

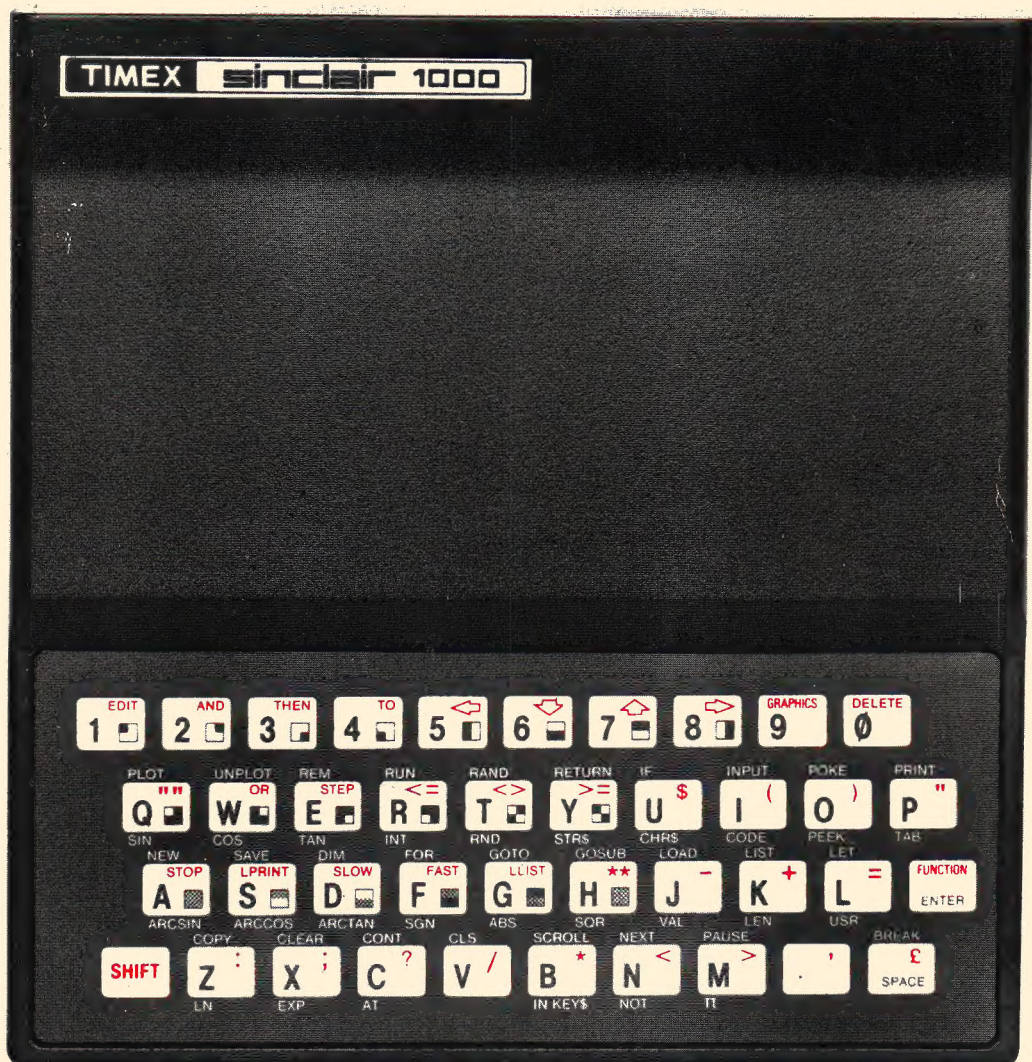
Mario Eisenbacher.

Programming

Your

Timex/Sinclair

1000TM in BASIC



JOHN HANLEY

Mario Eisenbacher received a B.S. from Rensselaer Polytechnic Institute and an MBA from Widener College. He is currently a project engineer at Westinghouse Electric's Combustion Turbine Division in Philadelphia. He has also completed *Programming Your Commodore 64 in BASIC* to be published in Fall 1983 by Prentice-Hall, Inc.

Programming Your Timex/Sinclair 1000 in BASIC

Mario Eisenbacher



PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey 07632

Eisenbacher, Mario.

Programming your Timex/Sinclair 1000 in BASIC.

"A Spectrum Book."

Includes index.

1. Timex 1000 (Computer)—Programming. 2. Basic
(Computer program language). I. Title. II. Title.
Programming your Timex/Sinclair One Thousand in BASIC.
QA76.8.T48E37 1983 001.64'24 83-3210
ISBN 0-13-729871-4
ISBN 0-13-729863-3

This book is available at a special discount when ordered in bulk quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

©1983 by Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

A SPECTRUM BOOK

All rights reserved. No part of this book may be reproduced in any form or by any means without permission in writing from the publisher.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

ISBN 0-13-729863-3 {PBK.}

ISBN 0-13-729871-4

Editorial/production supervision by Cyndy Lyle Rymer
Manufacturing buyer Christine Johnston
Cover design by Hal Siegel

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall of Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

Dedicated to:

My daughter
Alasan Elizabeth

My wife
Shirl

My sister
Yole Patterson

CONTENTS

Preface, xiii

INTRODUCTION, 1

Line Number Statement Function, 2

THE SEVEN LEVELS, 3

LEVEL ONE

YOUR FIRST PROGRAM, 6

Vocabulary, 6

YOUR FIRST PROGRAM, 7

INPUTS AND VARIABLES, 15

MORE PUNCTUATION: SEMICOLON, 21

CONTROLLING THE PRINT DISPLAY, 22

SUMMARY REVIEW, 27

INTEREST STIMULATORS, 27

EXERCISES, 28

LEVEL TWO

CREATIVE PROGRAMS USING LOOPS AND DECISIONS, 31

A “Key” Refresher, 31

VOCABULARY, 32

COMMANDS, 32

SAMPLE PROGRAM, 33

MATH SYMBOLS—THE NUMBERS GAME, 33

NUMERIC VARIABLES, 36

LOOPS, 42

FUN LOOPS, 45

HOW TO WRITE A PROGRAM, 48

SUMMARY REVIEW, 56

INTEREST STIMULATORS, 57

EXERCISES, 58

LEVEL THREE

HARDWARE AND MEMORY, 61

COMPUTER TERMINOLOGY, 61

Hardware, 61

Byte Consumption, 63

SOFTWARE, 64

EXERCISES, 66

LEVEL FOUR

CURVES, SHAPES, AND ARRAYS, 67

RANDOMIZING, 67

TRIGONOMETRIC FUNCTIONS, 69

OTHER FUNCTIONS, 72

GRAPHICS SYMBOLS, 73

TIME CONTROL, 76

OTHER USEFUL TOOLS, 77

A GUESSING GAME, 79

DRAWING A BAR CHART, 80

SOLUTION, 80

ARRAYS, 82

STORING AND RETRIEVING DATA, 85

INTEREST STIMULATOR, 89

SUMMARY REVIEW, 90

EXERCISES, 90

LEVEL FIVE

STRINGS AND LOGIC, 92

STRING FUNCTIONS, 93

SUBSTRINGS, 96

LOGIC, 99

BOOLEAN OPERATORS, 101

LOGIC AND LET, 104

LOGIC AND THE PRINT STATEMENT, 106

SAMPLE PROGRAM, 108

SUMMARY REVIEW, 115

EXERCISES, 115

LEVEL SIX

PEEKING, POKING, AND GAMING, 118

OTHER KEYS, 118

COMMUNICATING IN BASIC, 119

MEMORY STRUCTURES, 120

PEEK-A-BOO, 124

POKING AROUND, 127

HANGMAN, 130

NUMEROLOGY, 143

GUESS ME, 144

LET'S DRAW, 145

CATCH 'EM, 146

SLOT MACHINE, 147

KNOCKOUT, 148

EXERCISES, 150

LEVEL SEVEN

PRACTICAL APPLICATIONS, 151

GOSUB RETURN, 151

THE LOTTERY NUMBER, 152

SORTING NUMBERS AND WORDS, 154

INVESTMENT RETURN, 156
MORTGAGE LOANS, 157
INCOME AVERAGING, 159
THE END AND BEGINNING, 162

EPILOGUE: HISTORY OF THE TIMEX/SINCLAIR 1000, 164

Appendices, 167

Index, 187

PREFACE

Congratulations! You have in your hands the best instruction manual for computer BASIC known—and you probably have the most fantastic minisized maxipowered computer to go with it as well—the Timex/Sinclair 1000, brain-child of Clive Sinclair.

Of course, any computer—no matter how versatile—is merely electronic chips, wire, and molded plastic waiting for *your* instructions. Without you it is nothing; but with you it is a powerful tool. It can organize information, display data, play games, make music, teach courses, operate equipment, and do whatever else humans can think of. It has only one hitch—it speaks only one language, called BASIC. That is what this book is all about—to teach you the language your computer can interpret.

Studying a language would be fruitless if it were not used. Therefore, this manual is designed to be used with your Timex computer while you study.

Although the TRS-80, Atari, IBM, Texas Instruments, VIC, and Apple computers also “speak” BASIC, each has its own dialect. Once BASIC is learned, however, it is easy to convert to other dialects.

The Timex unit is an ideal unit to learn with. Besides requiring a minimum investment, it has sufficient “memory” (capacity to store and hold information) to more than satisfy your personal needs.

This manual was written with readers of all ages in mind. Although primarily designed for those with no computer knowledge whatsoever, those expert in BASIC will find it a convenient stepping-stone to the intermediate and advanced Timex-dialect BASIC manuals.

I hope you enjoy learning BASIC with your Timex. You will be able to write your own programs for profit or at the very least understand what goes into a program so you can make an intelligent decision on which prerecorded programs you should buy.

In any case—have *fun*!

Special acknowledgments to John Colarusso for “programming” me on the right track!

PROGRAMMING YOUR TIMEX/SINCLAIR 1000 IN BASIC

INTRODUCTION

You are about to embark on a language-usage course! The language is one called BASIC, which the computer can interpret.

BASIC (an acronym for Beginner's All-purpose Symbolic Instruction Code) was created in the late 1960s by Dr. John G. Kemeny at Dartmouth College, and was chosen over other languages (such as Pascal, COBOL, and FORTRAN) for its simplicity and adaptability to the types of uses of the small to medium-size computer.

All languages have two major characteristics—vocabulary and syntax (organization of words and symbols)—and so does BASIC. Your computer will respond only to certain words spelled in a specific way. Of the total BASIC vocabulary the Timex/Sinclair 1000 understands about 200 words, symbols, and numbers—a reasonable amount to learn. Of these, a dozen or so will enable you to write many programs (a list of instructions to the computer to perform specific tasks) before you finish Level One.

But note: How you use these words and symbols is important. This is the language's *syntax*. Computers only allow specific usages and ordering of vocabulary and punctuation—just as with the English language, where you learned that a noun requires a verb to complete a sentence. Instead of nouns, verbs, and objects your computer works with statements, commands, and functions.

Statements are individually defined and listed in your Timex manual—these are the words or phrases written in a program which define what the computer is to do. *Commands* are the same as statements in that they define a process. However, the command need not be a part of a program; it can instruct the computer what to do with an entire program. *Functions* are special words which are, in a sense, miniprograms stored in the computer you can use as needed. You will better understand these differences as they are introduced.

In order to maintain a logical arrangement of instructions to the computer, each begins with a *line number*. With your Timex, line numbers can range from 1 to 9999. It is good practice to start with number 10 and continue in steps of 10 (10, 20, 30, . . . , etc.). In this way you can always insert instructions between these lines (since the computer responds to instructions in numerical sequence) by using 11, 12, . . . , etc. The general format of an instruction then is:

Line number	Statement	Function
-------------	-----------	----------

Note that functions are not always part of an instruction, but when they are, they are positioned after the statement.

Punctuation in BASIC is very important. As in English, you use the comma, semicolon, and quotation marks; however, with the computer these seemingly insignificant little marks can have a significant if not drastic impact on your results. So pay particular attention to them as they are introduced. They are more powerful now than when you met them in grade school!

Keep in mind that the computer is essentially a mechanical device which operates only under specific commands. It responds (sometimes in a way you don't expect) to everything it sees (what you put in). Every comma and space becomes important. A computer is "dumb" in that it only does what it is told—no more, no less; if instructed to do something it cannot do, the T/S 10000 will simply stop. A computer's value is that it will do whatever you ask with speed and precision. Hence it is important you pay particular attention to exactly what you are instructing your machine to do.

Although there is a strict relationship between syntax and computer response, this in no way limits the computer's versatility. You will find that there are many ways to write a program to do the same thing—some only more efficient than others.

The Timex computer is unique in its relationship to BASIC. As with all computers, it responds to a dialect of BASIC slightly different from every other computer (but is readily translated). However, it has added shortcuts and techniques which allow it to pack much into a very small space. For example, it rarely allows you to write a program with the wrong syntax. It catches your typing errors and usually lets you know what they are.

For example, suppose you forget a punctuation mark when it is needed. Your line as typed will be prevented from being accepted by your T/S 10000 as a valid instruction; a special indicator called a *cursor* will appear showing you where you went wrong.

All the function and statement words are typed in, not letter by letter, but in one keystroke. This saves both time and memory space. The computer also has provisions for correcting your errors or making changes as you go along.

Most keys have more than one meaning. Some are automatically defined, depending upon the language syntax; that is, at given times depressing a particular key will only have a particular value. Some keys are not automatic and require you to shift (as on a typewriter) and depress a special key that puts the other keys in a special *mode* or condition where the keys will mean something else.

Before continuing with this manual please become familiar with the keyboard layout. Notice that the keys are arranged for the most part like those on a typewriter. Notice also the words and symbols above and below the keys as well as the symbols on the bottom right of the keys. On the top right of the keys you also find words or symbols in red. Also read the introduction and first chapter of your *User Manual* which came with your computer. This will best prepare you for your study of BASIC. We will be concentrating on developing your understanding of BASIC to a level where you can write programs on your own.

Programming Your Timex/Sinclair 1000 in BASIC is written with you in mind. Whether you are a novice or expert, the learning steps are identical—only the speed of grasping the syntax will differ. Hence this manual is written in *levels* rather than chapters. Each level constitutes an achievement in programming ability. It is similar to graduating from grade to grade in school. Once a graduate of Timex BASIC, there's no stopping you!

The secret to learning a language is *PRACTICE*. The secret to learning BASIC is practice *writing programs*. In the early levels we lead you through programs step by step while you *do it* on your *Timex*. You are encouraged to change programs yourself. If at any time you wonder, "What if. . .?" don't dismiss the thought: Try it and discover the answer to your question immediately. This is the best way to learn!

Each practice program—both in the text and as exercises—is designed to teach techniques in the *art* of programming. Yes, programming is an art as well as a science. Isn't writing in the English language an art? A well-written program is more than just the application of rules; it includes ingenuity and creativity as well. So if an exercise seems dull or you are just itching to try something terrific—be patient. Each step is a building block for a strong foundation. Just to wet your appetite we have, however, included short, simple programs called *interest stimulators*. These programs include statements and functions you will learn in a later level. As you progress, look back: You will be amazed that what once seemed indecipherable has become simple.

THE SEVEN LEVELS

Level One: This level is designed to give you a feeling of control over your computer. You will be able to manipulate what is printed on the TV screen. You will see your *name* on television! How about that!

Level Two: Now that you are in control you can write a program to do something besides just displaying words on the screen. A real accomplishment! You will be proud of yourself.

Level Three: It's time for a rest. Many textbooks start off discussing the history of computers and how they work. You are probably grateful not to have to sift through all that. But by now you can relate to the T/S 10000 and are beginning to wonder what that black box really is. You have no doubt told someone what you are learning. And they probably threw words at you like RAM, byte, chip, and so forth. You would feel more comfortable if you knew how these words fit in with what you are doing. Besides, your fingers are weary and you want something to read in bed. Go ahead! Relax a little—you will need it for Level Four.

Level Four: Now's the time to deal with numbers and with pictures (called *graphics* in computer jargon), including curves and graphs. You never realized this tiny computer could be so versatile!

Level Five: Here's where you discover a computer can also work with words—in a strange way. You also discover how logical your T/S really is. Using logic concepts, your programs can now include making decisions.

Level Six: Yes! You will have finally made it! You know almost every key on the Timex now. The fun really gets going! Several game programs are developed, including "Hangman." Think what else you can do with these programs after all you have learned!

Level Seven: Cleanup time! Also reality time—just in case someone asks you, "what can you do besides play games?" You will be able to demonstrate some practical programs!

Beyond Level Seven: It's up to you! An Epilogue on the history of the T/S 10000 is included as casual reading to give you an insight in the development of the T/S 10000. You will also find the appendices and index helpful. The appendices include answers to all the exercises as well as a chart format and listing of *error codes* (responses from the computer when an error is made in a program).

Although the levels are fluid in structure in accordance with increased levels of understanding, there is a general structured format for them all. Most levels include the following:

1. Definitions of new vocabulary and explanations of syntax
2. A step-by-step sample program using the newly introduced vocabulary
3. An interest stimulator—a short interesting program for you to use
4. Practice programs
5. Exercises (answers are in Appendix A), including programs similar to those in the text as well as totally new ones
6. Examples of errors and how to prevent or solve them
7. Summary of what you have learned

The best way to learn BASIC is to practice the problems with your computer as you follow along in the text. If you wonder what will happen if you do this or that, do not hesitate to find out by trying it. Do all the exercises: They are designed to enhance your learning experience.

Good luck!

LEVEL ONE

YOUR FIRST PROGRAM

Get ready, get set, go! Turn it on! If your computer is properly connected, you will see a blank screen with a letter "K" in the bottom left-hand corner. Note that the letter **K** is white on a black background. This is called the **K** cursor. Cursors are most helpful. They appear on the screen to let you know what the computer is waiting for. In this case the cursor indicates that the T/S is waiting for you to depress one of the keys which has a word above it, or a line number. More on this as we use the T/S in creating our first program.

Vocabulary

In this level you will become familiar with the following new terms and associated syntax (rules of usage):

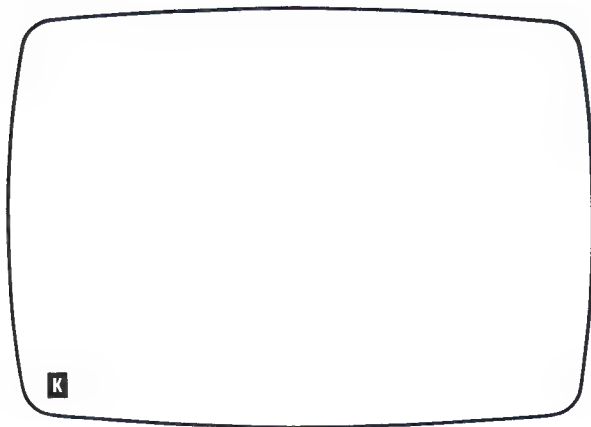
Commands: **ENTER, LIST, EDIT, RUN, GO TO, SHIFT, DELETE**

Keywords: **PRINT, INPUT, REM, CLS, NEW**

Functions: **AT, TAB**

Symbols: **Quotes (``), semicolons (;), comma (,), SPACE (#).**

FIGURE 1.1



Note: Whenever a **SPACE** is required in a program we shall use the “number” symbol (#) within the program. This will not be a problem since there is no “number” key on the T/S keyboard. This is a standard notation among all Sinclair users.

Other: String variables

Note: Before proceeding, verify you have a blank screen with only the **K** cursor on it. If other things appear on the screen, unplug the T/S by pulling out the power cable and then reconnecting it. This should clear it up.

YOUR FIRST PROGRAM

Type the following:

```
PRINT "TYPE NAME"
```

Notice that your cursor is no longer a **K** but an **L**. “L” stands for “letter,” meaning the single letter or symbol on a key. You will also observe that the cursor is always one step ahead of you—waiting!

Rule: Keywords cannot be typed in letter by letter.

Your display probably looks like this now:

```
PRINT RINT"TYPE NAME"
```

What happened? You did not realize that one key could display an entire word! So now you are wondering how you can correct this mistake without turning off the computer. Well, what do you want the computer to do? You want it to remove the extra letters. The common jargon is to "delete." Look on your keyboard and you will find the word **DELETE** in red on the number zero key. That is your rescue! Since the word is shown in red you are to depress the key labeled **SHIFT** first. While holding it down depress the **DELETE** key. One of your letters disappeared! Holding down the **SHIFT**, depress the **DELETE** again, and again. Don't stop. What happened when you came to the word **PRINT**? It disappeared all at once, didn't it? That is because it was a keyword and not individual letters. In trying to type this instruction you may encounter some difficulty until you are used to the nature of your Timex.

If you try to depress the letters P, R, I, N, T to spell **PRINT**, you will get **PRINT RINT**. The **K** cursor is more than a mere indicator; it also defines the mode you are in. In this case it is called the Keyword mode. This means the computer will only display keywords when depressing a key. Keywords are defined above each key and in this manual as well. You will soon become accustomed to the idea that one key will display an entire word and automatically include proper spacing. Can you see how this will save time and minimize errors?

So when you depressed the **P** key the computer automatically displayed the word **PRINT** with a space before and after. Then when you depressed **R** it displayed the letter only. Why? Why did it not show the keyword **RUN** shown above the letter **R** key? The reason for this is that once a keyword has been accepted by the computer it no longer "looks" for a keyword. Did you see the cursor immediately change to **L**? It is now looking for a letter!

Guideline: Keywords will be represented throughout this manual in **boldface** type. The number zero will be distinguished from the letter "oh" by a slash through the number: Ø. (Printers used with computers also make this distinction.)

Since you have "deleted" the word **PRINT**, start all over, remembering not to type the individual letters.

Rule: When instructing the computer to **PRINT** a word, the word must be presented in the instruction within quotes.

You will note quotation marks appear both on the **Q** and **P** keys. Do not use the ones on the **Q** key; their use will be explained later. Only the symbol on the **P** key will do what we want for now.

Reminder: If you make an error as we go along, use **DELETE** to correct it.

The quote on the **P** key is used at both ends of a word or phrase. To obtain the quote symbol you must be in the Letter mode, i.e., with the **L** cursor. Then depress **P** and hold **SHIFT**.

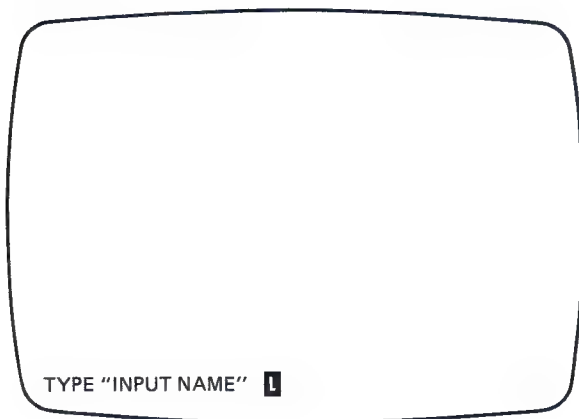
Rule: Any symbol in the upper right-hand corner of a key is obtained by using the **SHIFT** key only when you have an **L** cursor on the screen.

Rule: All words above the keys are statements or commands and are directly obtained when in the **K** or Keyword mode.

When you typed, words appeared on the *bottom* of the screen—this is called the “working area.”

Note: The T/S 1000 can display 32 characters in a line and 24 lines. The bottom 2 lines are used as work space before being put into a program.

FIGURE 1.2

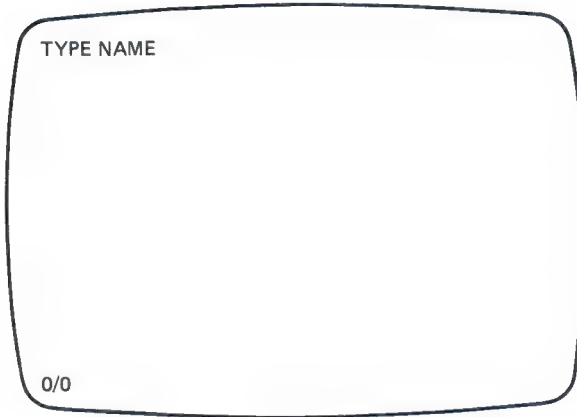


Congratulations! You should now have and are ready to release your command to the computer. This is done by depressing the **ENTER** key. **ENTER** instructs the computer to look at what you put in and act upon it.

Well, what happened? You gave the T/S 1000 the command to **PRINT** whatever is in quotes. So that's what it did at the top of the screen. Note the

bottom left-hand corner has 0/0. This is the error code. The first digit defines the kind of error and the second defines the location. Two zeros (as here) mean the program worked properly.

FIGURE 1.3



The error code is always displayed after you **ENTER** a direct command or after the T/S has completed its set of instructions (the program).

So far you haven't written a program—only a command to display a sentence. But before we start a program, experiment with the **PRINT** keyword and quotes. Practice on your own: Put anything into quotes you wish. Experiment with all the keys. See what happens when you depress **ENTER**. Try it without one of the quote symbols. What happened? An error of some kind! It won't list!

Reminder: The best way to learn a language is by practicing.

When you feel comfortable with the **PRINT** command depress **CLS** on the **V** key. This is the instruction to "clear the screen." Then depress **ENTER**. This releases the instruction to the computer telling it to clear the screen. Clearing the screen means to remove from the display everything on the screen except the cursor or error code. Don't worry; you will learn next how to keep what you typed in even when the screen is cleared.

It is good practice between programs to insure that there are no old instructions in the computer which might conflict with your new instructions. This is accomplished by giving the command **NEW** followed by **ENTER**. **NEW** is a keyword over the **A** key. Although this removes all your instructions inside the computer, don't panic—you will learn later how to keep programs forever by taking them out of the computer and putting them in a safe place.

Now depress **NEW** and **ENTER**. Then type the following by first depressing numbers 1 and 0.

10 PRINT "TYPE NAME"

The **K** cursor will accept up to four digits because it is looking for a keyword and/or line number. In this case the line number is 10. Still in the **K** mode the computer is looking for another digit or a keyword. When you depressed the **P** the word **PRINT** was displayed.

Did you try to put a gap **SPACE** after the 10? It was not necessary because the keyword **PRINT** consists of seven characters: **#PRINT#**, where the two spaces represented as **"#"** are invisible.

Did you try to put in a space between the words **TYPE** and **NAME**? What happens if you leave the space out? Try it. It reads **TYPENAME**—a little difficult to read, isn't it?

Rule: The computer displays exactly what is between the quotes in a **PRINT** statement including (or excluding) spaces.

Note: As a matter of convention the "number" symbol (**#**) will only be used where the **SPACE** key is required and not where obvious, as in a sentence.

Depress **ENTER**. What happened? Line 10 moved up to the top of the screen intact. The **K** cursor is at the bottom waiting for more instructions. Go ahead, type in another line:

```
20 PRINT "      "
```

and fill in the quotes with whatever you want. Type in a whole paragraph if you wish—your Timex can display a lot. Time to let it go to the top of the screen. Depress **ENTER**.

Now let's type in something wrong to see how the Timex handles syntax errors: First depress **NEW** and **ENTER**. Then:

```
10 PRINT "HELLO
```

leaving out the second quote mark. Depress **ENTER**. What happened? Why didn't it go to the top of the screen? Did you notice a strange symbol appearing at the end of **HELLO**? It is a white **S** on a black background known as the **S** cursor or Syntax cursor. It is located exactly where the error is; note the **L** cursor is still on the screen. Now type in the quote; watch the **S** disappear and the **L** move over. **ENTER** your line.

Let's try for another error. First depress **NEW** and **ENTER**. Now type

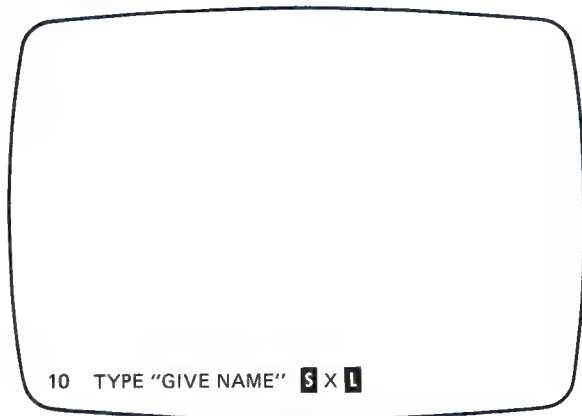
```
10 PRINT "HELLO"X
ENTER
```

12 Your First Program

You discover another syntax error. You have something that is improperly presented. The **S** is between **HELLO** and **X**:

```
10 PRINT "HELLOSXL"
```

FIGURE 1.4



We could insert punctuation, but we shall assume the **X** is unwanted. So since the **L** cursor is to the right of the **X** when you depress **DELETE** (use the **SHIFT** key), the **X** should disappear. Try it. What happened? Right! When you got rid of the **X** the **S** cursor automatically disappeared as well! Why? Because the *reason* it was there disappeared when the **X** was removed. Now **ENTER** line 10.

Let's try one more form of correction. What if you typed in a complete line and you discovered an error in the middle? Is it necessary to **DELETE** everything back to that point? No, it isn't.

First depress **NEW** followed by **ENTER**.

Now type in


```
10 PRINT "TODAY WAS FRIDAY"
```

Include the natural spacing between words within the quotes, as you do with a typewriter. Do not **ENTER** this instruction. You suddenly realize you meant to type "TODAY IS FRIDAY". How do you change **WAS** to **IS**? Easy! Move the **L** cursor to the left. How? Hold down the **SHIFT** and depress the left *arrow* key on the numeral 5 key until the **L** is between the **A** and **S** in the word **WAS**. If you went too far you can move the cursor to the right using the **SHIFT** and 8 key in the same manner. Now depress **DELETE**—using **SHIFT**, of course. Do it twice, erasing the **W** and **A**. Now depress the **I** key. Yes, it is correct; although the **L** cursor is sitting in there, it will automatically disappear when you depress **ENTER**. Try it!

This is what you now have at the top of the screen:

```
10 PRINT "TODAY IS FRIDAY"
```

and a **K** cursor at the bottom waiting for the next instruction.

That arrow next to the line number is a white arrow on a black background . It is called an *Arrow cursor*. It cannot move left and right. It can, however, move up and down when you have several lines displayed. Its use will be described later.

Reminder: Words which require a single key action instead of typing letters are in **boldface** type.

How do we make corrections to a program once it is listed—i.e., at the top of the screen? It is a process called *editing*—proofreading and making changes. This procedure will be described later. It is sufficient right now to know of its existence.

Well, so far, so good! You are learning how to manipulate characters and how to deal with cursors. Other cursors will be introduced later over which you will have full control.

What if you want to delete a whole line? Do you have to keep depressing the **DELETE** key? Thankfully, no!

Rule: To delete an entire line already listed at the top of the *screen*, type onto the screen the same line number and (without any key-words) depress **ENTER**.

Try the above rule and watch as an entire line disappears from your program.

Reminder: Line numbers are needed in a program to organize and identify a set of instructions. They must all be different and in the range 1 to 9999.

Hint: Although you can use line numbers 1, 2, etc., it will prove frustrating unless you use numbers in 10s: 10, 20, 30, 40, etc., or 110, 120, etc. This allows room within the program to insert lines as an afterthought.

Before proceeding, be safe—use **NEW** and **ENTER**.
Now type:

```
10 PRINT "TYPE NAME"  
20 PRINT "TYPE STATE, ZIP"
```

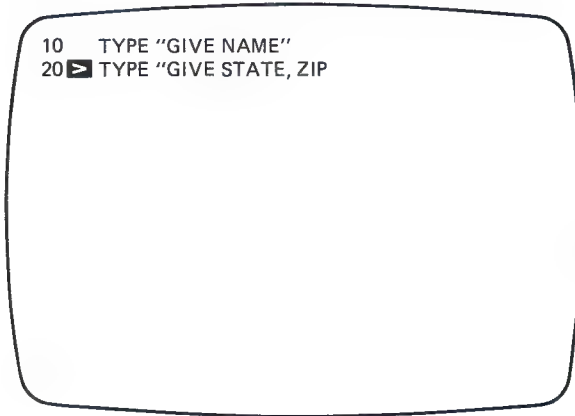

14 Your First Program

Depress **ENTER** after each line is completed.

Rule: You can put any symbol or character within quotes. The computer will print exactly as listed within quotes.

You now have two statements at the top of the screen. Notice the arrow cursor moved to line 20—representing the last line created.

FIGURE 1.5



Now type:

30 **PRINT** "TYPE CITY" and **ENTER**

40 **PRINT** "TYPE STREET ADDRESS" and **ENTER**

Rule: Whenever a line is completed at the bottom of the screen work space, depressing **ENTER** puts the line into the program.

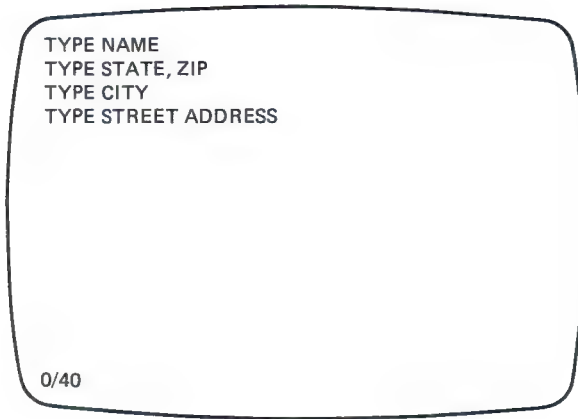
Instructions can be as long as necessary—the computer will simply display on the next line automatically.

So far you have created a program of four lines. Let's see what it does. To find out we must give the instruction or command to the computer to follow all instructions given. Those which have line numbers are called *lists*. The instruction to compute is **RUN**. This command is also a keyword and is directly accessed by depressing the **R** key.

Depress **RUN** (R key) and **ENTER**

Well, it did what it was told—it printed the four statements in quotes, one after the other. The error code 0/40 indicates the computer was able to follow the instructions and the program completed line 40.

FIGURE 1.6



This program doesn't really do much—yet. We would like to be able to actually type in a real name and address and have the computer display it in an orderly manner.

First let's call our program back from inside the T/S 1000. How? Depress **LIST** (K key) and **ENTER**

Lo and behold—our program appears just as it was before.

Definition: **LIST** is a command that calls for a listing of the program.

The arrow cursor is at line 0 above the screen and can be called down by depressing **SHIFT** and the 6 key. This cursor will in no way affect the listing or the operation of the program.

Definition: **RUN** is a command that tells the computer to follow the instructions of the program you keyed in.

Before we can put information called data into the computer we must learn a new term: *variables*.

INPUT AND VARIABLES

The sentences in the **PRINT** statements you have programmed imply that additional information can be provided by you, the programmer. Thus "TYPE NAME" implies you are to type your name. But how do we get the computer to accept your actual name? This is done using an **INPUT** statement and a variable.

You want the T/S 1000 to display the sentence in line 10 asking for information or data. Then you want the Timex to do something with it. So type in:

15 INPUT N\$

(\$ is on the U key) and

ENTER

Note line 15 automatically slipped into its sequential location.

Definition: **INPUT** is a keyword on the I key; it is a command which causes the computer to stop and wait until you enter a character (number, letter, or symbol). **INPUT** can only be used within a program listing.

The **INPUT** statement must be followed by a name identifying data. This name is called a *variable* since its value or what it represents could vary or change.

There are two types of variables in Timex BASIC: numeric variables and string variables. *Numeric variables* represent real numbers. *String variables* represent a "string" of characters (numbers, letters, and/or symbols). Since we are dealing with names and addresses which are strings of characters, we are to name them in accordance with the syntax rule for identifying string variables; therefore, **N\$**.

Rule: String variables must be identified by a single letter followed by the "dollar" sign character (**SHIFTU**).

"N" was selected because the word **NAME** begins with N: it makes the program easier to understand.

Now continue typing with:


25 INPUT S\$ ENTER

35 INPUT C\$ ENTER

45 INPUT A\$ ENTER

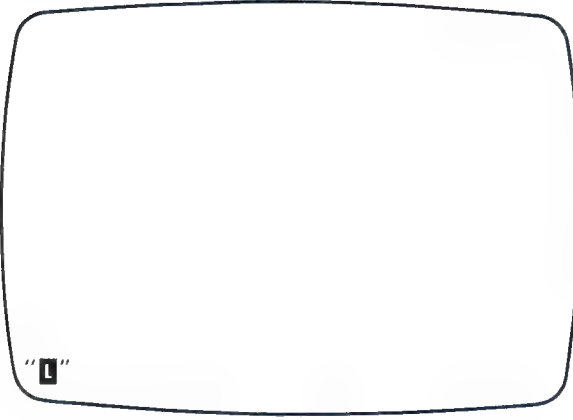
By now you must realize the **ENTER** key is always used when a program line is to be listed. It will not be shown in the text as a separate step henceforth.

Now **RUN** the program (depress **RUN** then **ENTER**).

Notice the  cursor at the bottom of the screen is in between quotes. These quotes were provided automatically because the input statement asked for a string variable. You can insert any character you wish.

Respond to the sentence at the top of the screen—i.e., type your name—and the cursor will expand the quotes to accept whatever you type in no matter the length. When you are finished typing depress **ENTER**. Your response disap-

FIGURE 1.7



peared, and the second sentence appears at the top. Now type in your state as requested and **ENTER**.

Again, respond likewise to the third and fourth sentence. After your last input the error code is displayed as 0/45.

Well, this inputting of data doesn't appear to do much; but your computer does have added information now. After all, the values of the variables **N\$**, **S\$**, **C\$**, and **A\$** are stored somewhere in the T/S 1000 just as you typed them in.

Just to show you this, type **PRINT N\$** (followed with **ENTER**, of course). See—it printed your name! It does remember!

OK—let us proceed to do something useful with the data the T/S now has. First, call back the program:

LIST ENTER

Add a statement to print the variables in succession:

```
50 PRINT N$, S$, C$, A$
```

Depress **RUN**.

Provide responses to the requests for data.

Note: When you **RUN** a program, all variables (string and numeric) are reset to 0 or empty. This means each time your program is **RUN** you must reinput the data.

Results? The screen displays the values of the variables (as you provided) in a peculiar format.

FIGURE 1.8

```
TYPE NAME
TYPE STATE, ZIP
TYPE CITY
TYPE STREET ADDRESS
YOUR NAME      YOUR STATE, ZIP
YOUR CITY      YOUR ADDRESS

0/50
```

You had full control over the format of the display through the use of those commas in line 50. You also have control of when and what is actually displayed. Since it is unnecessary to show the instructions on the screen simultaneously with the responses, let's clear the screen after all the variables are given but before the name and address are displayed. So

LIST

and add line 47:

47 CLS

Definition: **CLS** is a command which will clear the screen only; it does not affect any variables.

Also, let's put the information in normal order—name, street, city, state, and zip, or N\$, A\$, C\$, S\$:

50 PRINT N\$, A\$, C\$, S\$

Now **RUN** the program. Only the responses appear, though probably not very neatly.

Now is a good time to learn how you have control over what you display on the screen using punctuation.

In BASIC a *comma* is used to instruct the computer to begin the next display at the next column. The T/S 1000 is factory-programmed to have a two-

column structure each with 16 spaces (32 per line). Hence the first variable or string starts at the first space; the second will start at the sixteenth space.

Hint: A series of variables with commas separating them will continue to print in a two-column format, automatically shifting to the next line since the instruction does not allow a backward motion. This results in an orderly display.

Now depress **LIST**

Your program should be (note if you took a break or turned off your unit—retype it in—later you will learn how to properly keep it on cassette tape):

```
10 PRINT "NAME?"
15 INPUT N$
20 PRINT "STATE AND ZIP?"
25 INPUT S$
30 PRINT "CITY?"
35 INPUT C$
40 PRINT "STREET?"
45 INPUT A$
47 CLS
50 PRINT N$, A$, C$, S$
```

RUN the program, but as you respond to the sentences limit your data to less than 15 characters for now.

The display should be:

```
Your name   Your address
Your city   Your state ZIP
```

Let's experiment with the **PRINT** statement. Another way of writing the print is as follows:

```
50 PRINT N$,
52 PRINT A$,
54 PRINT C$,
56 PRINT S$,
```

Note the use of the commas. **ENTER** the above lines. (Note as soon as you **ENTER** the first line, the list will automatically display itself with the Arrow cursor pointing to the line you just typed.) If you feel more comfortable seeing the program while typing, **LIST** the program (followed by **ENTER** of course); then type in line 50, etc.

FIGURE 1.9

YOUR NAME YOUR CITY	YOUR ADDRESS YOUR STATE, ZIP
0/50	

Reminder: Depress **ENTER** after a command. *Stop!* Don't depress **RUN**. Remember the rule that **RUN** clears all variables—that means you will have to type in your name and address all over again.

Hint: Save your variables: Instead of using **RUN**, type in the command **GO TO 47**. This directs the computer to start at line 47. Your program will otherwise operate as normal. (**GO TO** is a keyword on the **G** key.) This command will be reviewed in detail later; it is only presented here to save you unnecessary work.

Verify that your responses are still in the T/S by **GO TO** and then type

51 PRINT

followed by

GO TO 47

What did this do? It said "go to the next line." A **PRINT** statement alone is like a carriage return on a typewriter, and gets you to the next line.

Your display has now been reorganized and reads:

Your name

Your address Your city

Your state and ZIP

Now redo lines 50, 52, 54, and 56 by putting two commas in each, such as

```
52 PRINT N$,,
```

and remove line 51. Your result is neatly displayed as

```
Your name
Your address
Your city
Your state and ZIP
```

Why did this work? Remember that the comma directs the display to the next column. Two commas in succession instructs the computer to display nothing at the second column. A subsequent comma sends the display to the next column. Do you see how you can manipulate the display with a comma?

Now go back and remove all the commas in lines 50, 52, 54, and 56. Your result will be the same as above. Why?

Remember, use **GO TO 47**. If you accidentally depressed **RUN**, you will have to type in the information again, since it will have been erased. You will learn later how we can retain information permanently.

Let's try to simplify the above display in one statement. First erase lines 52, 54, and 56 by typing each number followed by **ENTER**. (Remember how to delete an entire line?)

Then type

```
50 PRINT N$,, A$,, C$,, S$
GO TO 47
```

Note that combining all outputs in one **PRINT** statement requires some control punctuation such as the comma to separate the string variables and control their location on the screen. There are other punctuations available in your Timex which you will learn before Level Two.

Earlier you were asked to restrict your responses to 15 characters because the column width is 16. What do you think would have happened if you went over 16? Go back to try it and see!

Had you inputted a name or address composed of more than 16 letters, it would have affected the display. Since the sixteenth space would have been filled, the next **PRINT** instruction would have automatically moved to the next available position. Practice with the comma until you fully understand how it affects the display.

MORE PUNCTUATION: SEMICOLON

Getting back to your program listing, try the following:

```
50 PRINT N$; A$; C$; S$ (; is SHIFT X)
```

Type the command

GO TO 47

Rule: The *semicolon* in a **PRINT** statement causes immediate printing after the last character displayed. It does not include any spacing.

It could also be written in this manner:

```
50 PRINT N$;  
51 PRINT A$;  
52 PRINT C$;  
53 PRINT S$;
```

or

```
50 PRINT N$; A$;  
51 PRINT C$; S$
```

Well, do you think this would make a good address label? You can force some spaces by printing “#” (where # means depressing the space key).

LIST your program again. Type in the following **PRINT** statement:

```
50 PRINT N$; "#"; A$; "#"; C$; "#"; S$
```

followed by **GO TO 47**


Now it is a lot more legible. However, it still lacks something.

Hint: Two semicolons together do nothing for you.

Reason: Subsequent semicolons are merely repeating the same instruction—to look at the last character displayed on the screen, which is not affected by the many semicolons placed after it.

CONTROLLING THE PRINT DISPLAY

Your Timex has the ability to **PRINT** wherever you want. To do so involves two important functions: **TAB** and **AT**.

Although these are not functions in the strictly mathematical sense, they are treated as such by the Timex in that they are accessed using a special shifting key labeled **FUNCTION** and cursor depicted as a white  on a black background.

Since **TAB** and **AT** cannot be used by typing individual letters, they too will be presented within a program in this book in **boldface** to remind you to depress a single key.

TAB is found on the **P** key, while **AT** is found on the **C** key. To access these you must first enter the **FUNCTION** mode by getting the **F** cursor: Depress the **FUNCTION ENTER** key while holding down the **SHIFT**.

When the **F** cursor appears on the screen in place of the **L** or **K** cursors you can then depress any key (but only once) and obtain the function as written under each key. Immediately thereafter the cursor reverts to an **L** cursor.

To understand these functions consider the display screen. It consists of 24 rows of 32 characters each. The bottom 2 rows are reserved for work space. This leaves a screen capable of displaying 22 by 32 or 704 characters in all. Refer to the Video Display Chart in Appendix B.

Look at the chart and assume you are on line 10 and you wish to start a display at column 15. You then instruct the computer to start at **TAB 15**. Isn't this the way a typewriter works? Do you see how you could override the comma control of 2-column sections? You could use **TAB 4**, then **TAB 8**, etc. and create an 8-column spread!

Rule: **TAB** in a **PRINT** statement should always follow the keyword **PRINT**. **TAB** is always followed by a number between 0 and 31 and a semicolon. Thereafter the variable or quoted material is typed. More than one **TAB** is separated by semicolons.

As an example try this **PRINT** command (not using line numbers). Note that this will not affect the program presently in memory (storage), but also remember not to use **RUN** and erase all your variables!

```
PRINT TAB 5;1; TAB 10;2; TAB 15;3; TAB 20;4; TAB 28;5; TAB 35;6  
TAB 50;7
```

When typing don't worry about looking neat on the screen in the work space. Watch it on the screen as you type and observe the automatic spacing before and after each function keyword.

What did this do? Practice using words instead of numbers, such as **TAB 5; "HELLO"; TAB 10;"TODAY";** etc. Where did **TAB 50** put the 7? Experiment with this: Remove some semicolons or throw in a few commas and see what happens.

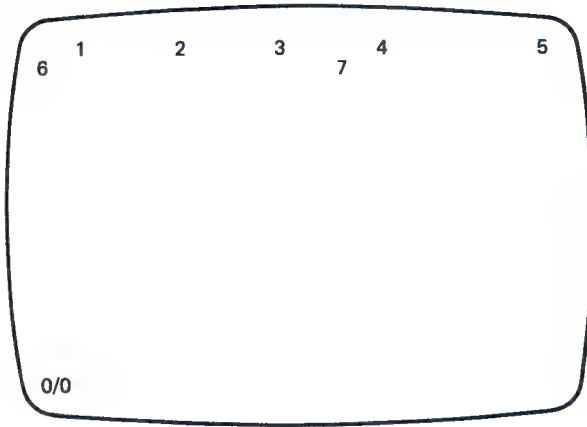
TAB 50 put the number on the second line. Since the computer cannot backspace, it simply goes to the next line. The statement

```
PRINT TAB 1; 1; TAB 31; 2; TAB 2; 3
```

will clearly show you what is meant about backspacing. Also, any number over 31 is automatically reduced by a factor of 32 and uses the remainder. So **TAB 65** is the same as **TAB 1**. (That is, 65 divided by 32 is 2 with a remainder of 1.)

Practice with **PRINT TAB** statements until you feel you have control. Now you are ready for the more sophisticated **AT** function.

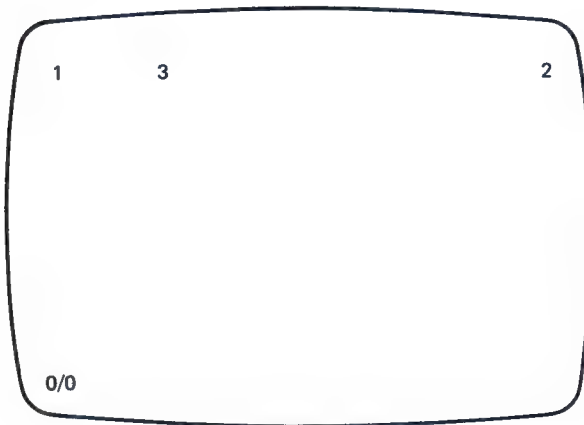
FIGURE 1.10



AT follows the same rules as **TAB**—the use of semicolons. **AT**, however, uses two numbers: the line (row) number and the column number separated by a comma. For example:

```
PRINT AT 5, 2; 1; AT 5, 31; 2; AT 5, 10; 3
```

FIGURE 1.11



(**AT** is on the **C** key.) You will notice that using the **AT** function you can “backspace.” Line numbers must be in the range 0 to 21 and columns 0 to 31. No other numbers are allowed. Try:

```
PRINT AT 40, 1; 1
```

You received error B. See Appendix C (B = “out of range”).

PRINT AT 5, 35; 1

This will also yield the same error. And finally, if you tried:

PRINT AT 5; 35; 1

you would get a syntax error because a comma must be between the line and column numbers.

Now, if you haven’t lost your old program accidentally (by turning the machine off, for example), type:

GO TO 47

If the old results are not there, retype the program as before. Then depress

LIST

Now let’s get more sophisticated with our **PRINT** statement based upon what we have learned so far.

Let’s try to center the display on the screen; try tab location line 8, column 5 to start the name. The street address should be directly underneath: We could use **AT 9, 5** or **TAB 5** or **TAB 37**.

For the city we could use **AT 10, 5** or **TAB 5** or **TAB 37** or **TAB 69**.

Let’s put the state directly after the city with a comma using simply “, #”; **\$\$**. Remember, “#” means depress **SPACE**. Note the comma is *within* the quotes and will cause no problems: It will simply be displayed as a character.

So let’s try:

**50 PRINT AT 8, 5; N\$; TAB 37; A\$; TAB 37;
C\$; “, #”; \$\$ ENTER**

followed by

GO TO 47

You should experiment with line 50 by changing it and see the results (remember to use **GO TO 47**). But first let us examine another time-saver.

Editing

Type the command

LIST 10 followed by **ENTER**

FIGURE 1.12



You will note line 10 automatically goes to the top of the screen with the Arrow cursor. Just plain **LIST** really means **LIST 0**.

Depress

LIST and ENTER

Where is the cursor? Off the screen! It went to line 0 automatically; using the downward arrow key will immediately bring it into view. You can move the Arrow cursor up and down using **SHIFT 6** and **7** keys (**5** and **8** keys have no effect).

You can move it to line 50 or to save time use the command **LIST 50**.

Now with the arrow at line 50, which is what you want to change, let's call line 50 to the bottom of the screen (into the work space).

Depress **EDIT** (obtained by **SHIFT 1**). The cursor changed to a **K**.

In the above example of line 50 let's say you want to change the second **TAB 37** to **TAB 69**; to do this proceed as follows:

Move the cursor to the right by depressing the right arrow key (**SHIFT 8**). Notice that the cursor changed to an **L**.

Continue depressing until the cursor is just to the right of the 37 you want to replace. Then use **DELETE** (**SHIFT 0**) twice to remove 37. Then type in 69 followed by **ENTER**. Then type **GO TO 47**. Practice by editing other lines.

REMARKS

Let's add one more touch to your program:

```
5 REM "NAME AND ADDRESS"
```

```
ENTER
```

REM is a keyword and will automatically print after the line number when in **K** mode by depressing the **E** key.

It is now the first line in your program. You will find that it has no effect on your program whatsoever. **REM** is short for **REMARK**. Anything can be written after the **REM** statement. This only appears when you **LIST** the program. When you write long programs and wish to type in some notes or description of what is going on solely for information, you can use the **REM** statement.

Titles such as those above were put in quotes to get you into the habit of naming programs so they can be recognized when stored on cassette tape. If it has a name, the Timex can be instructed to find the program on the tape.

One more time, **LIST** your program. Review each line in your mind. Command **GO TO 47** and see your final result. Now **RUN** your program and input someone else's name and address.

You have now completed your first program! Although it is a simple one, it was designed to enable you to concentrate on rules and syntax.

SUMMARY REVIEW

Let's review what you have learned:

1. A new vocabulary including
**ENTER LIST EDIT RUN GO TO PRINT INPUT NEW
 REM CLS AT TAB DELETE SPACE**
2. New uses for the comma, semicolon, and dollar sign
3. How line numbers are used in a program
4. How string variables are used to input data
5. The use of cursors in Timex BASIC
6. How to manipulate the display on the screen using punctuation and functions
7. How to identify errors and correct them

INTEREST STIMULATORS

As promised, here are two challenging interest stimulators: You are on your own! To **RUN** or not to **RUN**—that is the question!

```
10 FOR I = 0 TO 63
20 PRINT TAB 4 * I; I;
30 NEXT I
RUN
```

28 Your First Program

Notes:

= is on **L** key
* is on **B** key
NEXT is on **N** key
FOR is on **F** key
TO is on **4** key

Now try:

```
10 FOR N = 0 TO 255
20 PRINT CHR$N;
30 NEXT N
RUN
```

Note: **CHR\$** is a function on the **U** key.

Hint: Clear the previous program using **NEW**.

Don't fret—by the end of Level Two you will understand the above two programs thoroughly!

EXERCISES

1.1 Make this statement correct and explain why it's wrong:

```
20 PRINT NAME, CITY
```

1.2 Make this statement correct:

```
20 INPUT N$, C$, S$
```

1.3 Can you use **REM** anywhere you want in a program?

1.4 What happens if you depress **LIST** when there is no program to list?

1.5 Is this legal in Timex BASIC?

```
RUN 30
```

- 1.6 How do these two print statements differ?

```
PRINT 1, 2, 3
```

```
PRINT 1; 2; 3
```

- 1.7 Write a **PRINT** statement to locate number 1 in Column 0, 2 in Column 6, and 3 in Column 18 without using **TAB** or **AT** functions. Follow this with a print statement using **TAB** function to prove it.
- 1.8 Do you need an **END** statement at the end of a program in Timex BASIC?
- 1.9 What is wrong with the following as it stands?

```
PRINT TAB 15; 4; AT 5, 5; 6; TAB 20; A$
```

- 1.10 What will be the result of the following program? (After you answer, try it.)

```
10 INPUT A$
```

```
20 INPUT B$
```

```
30 INPUT C$
```

```
40 PRINT AT 11, 15; B$
```

During the program **RUN**, type in the words **ME**, **I**, and **YOU** for

A\$, **B\$**, **C\$**

- 1.11 String variable data must be entered within a pair of quotation marks. True or false?
- 1.12 Is the following legitimate in a program?

```
130 PRINT
```

```
140 PRINT
```

```
150 PRINT
```

- 1.13 What is wrong here?

```
101020 PRINT "NAME"
```

- 1.14 Answer true or false: A literal is restricted to 16 characters. (A literal is anything written in quotation marks.)
- 1.15 Answer true or false: **CLS** will only clear the screen; variables remain intact.
- 1.16 Answer true or false: **RUN** will clear the screen and change all variables to 0, prior to printing.
- 1.17 What does this statement do?

```
50 PRINT "NAME", "ADDRESS"
```

- 1.18 Answer true or false: Literals can only be used in print statements.
- 1.19 You are in charge of an auto parts department in a large chain store. A customer wants to buy a muffler for his car. You find the part number to be 14378 and the price is \$17.50.

Write a program to display the repair part description, part number, and unit price. Base your program on the one presented in this chapter. Look at the answer only if you get stuck. Note that our answer is only one possibility. The best test for your program is to **RUN** it and see the results!

- 1.20 Write a unique program to display names of three people you know—their first name, middle initial, and last name each under a column heading. Results should be, for example:

FIRST	MIDDLE	LAST
JOHN	F	KENNEDY
TOM	G	BRYAN
HARRY	S	TRUMAN

Include statements in your program describing what you are doing.

(Answers are in Appendix A.)

LEVEL TWO

CREATIVE PROGRAMS USING LOOPS AND DECISIONS

Now that you have experience with your Timex and know simple BASIC vocabulary you are ready to learn some more practical uses for the computer.

Less time will be given to explanations of what you already know, and more to new vocabulary and syntax.

Concepts covered here include the powerful tools of looping, decision trees, counters, and numeric variables. In developing these concepts, we shall develop a practical computer program.

A "Key" Refresher

Note that almost every key has a letter or symbol similar to those on typewriter keys. All words above the keys are keywords and are keyed directly when (and only when) in the **K** mode. Words below the keys are functions and obtained directly after changing to the **F** mode. Words and symbols in red on the right-hand corner are obtained by holding the **SHIFT** key. The symbols in the lower right corner of some keys are obtained directly after getting into the **G** mode.

VOCABULARY

In this level you will learn the following new BASIC words:

Commands: **NEW CLEAR CLS CONT STOP**

Statements: **IF . . . THEN**
FOR . . . TO . . . STEP
NEXT
TO

Terms: loops, flowchart, decision tree, counter, graphics mode, for-next loop, conditional loop

COMMANDS

NEW (A Key)

Depress the command **NEW** (followed by **ENTER**) whenever you want to clear the computer program and variables. This has the same effect as turning the computer off by pulling the plug out and then plugging it back in.

CLEAR (X Key)

Use the **CLEAR** command if you wish to wipe out all variables in memory and everything on the screen. However, the listing is still intact in the computer's storage called *memory*.

CLS (V Key)

Use the **CLS** command if you only want to clear the screen. This is often used within a program to clear the screen after the display has been seen and is not needed.

CONT (C Key)

Use this command if your program stops because of lack of space on the screen. **CONT** will continue from the line number shown in the error code until the program is finished or more space is needed requiring another **CONT** (continue) statement.

STOP (SHIFT A)

Use this command within a program to make it stop (until you depress **CONT** to continue).

All the above commands can be typed directly to tell the computer to do something, or they can be used within a program with a line number to tell it to do something in the middle of something else.

SAMPLE PROGRAM

You are owner of a small business. You have three employees: John Green, Tom Jones, and Eileen Nelson. Their respective hourly wages, based on experience, are \$5.49, \$3.18, and \$8.42. This week John worked 35 hours, Tom put in 40 hours, and Eileen worked 38 hours.

You want to display a list of your employees showing their wage rate, hours worked, and gross pay. Also, you would like to know what the total payroll is.

To solve this problem with a program you will need to know how to use:

1. Math symbols
2. Numeric variables
3. **LET** statement
4. **IF . . . THEN** statements
5. **FOR . . . NEXT** statements and loops
6. Flowcharts

These concepts will be thoroughly developed before we attempt to write the program for the above problem.

MATH SYMBOLS—THE NUMBERS GAME

The usual four arithmetic functions are represented by the plus sign “+” (**SHIFT K** key), minus sign “-” (**SHIFT J**), multiplication sign “*” (**SHIFT B** key), and the division sign “/” (**SHIFT V** key). Note you cannot use the usual “X” for multiplication. Other symbols of immediate need are the equals sign “=” (**SHIFT L** key), the greater than sign “>” (**SHIFT M** key), and the less than “<” (**SHIFT N** key). Also of use will be the left parenthesis “(” and right parenthesis “)” on letter keys **I** and **O**.

When working with math, be especially careful not to confuse the number 0 with the letter O or number “1” with the letter “I”.

Any math problem can be directly solved by using the command **PRINT** without a line number:

PRINT 2 + 3

34 Creative Programs Using Loops and Decisions

When **ENTERED** you will see the answer 5 at the top of the screen.
Now try:

```
PRINT 7* 9/3 - 4* 3/2
```

What answer do you expect? The answer 15 is logically obtained and is the *only* possible answer as far as the computer is concerned. The Timex performs multiplication and division in order of appearance. Then it performs addition and subtraction in order of appearance.

In the above example here's what your computer did: In the sequence of mathematical operations

$$7 * 9 / 3 - 4 * 3 / 2$$

the $7 * 9$ was computed first as 63. It has now become

$$63 / 3 - 4 * 3 / 2$$

and computes $63/3$ or 21 and the problem is reduced to

$$21 - 4 * 3 / 2$$

Following the rules of order, the next computation is $4*3$ because it is the nearest multiplication or division to the most recent computation. The subtraction must wait until all multiplication and division are completed.

With $4*3$ equal to 12 the problem is reduced to

$$21 - 12 / 2$$

From here, can you see how we get 15?

This can be stated more clearly if you use parentheses:

```
PRINT((7*9)/3)-(4*3)/2)
```

will give the identical result. Try it! But make sure you get in all the parentheses. A right parenthesis without the left parenthesis (or vice versa) is an error of syntax!

With parentheses you can override the computer's natural logic and force it to perform in the order you want. Try this with the same numbers:

```
PRINT (7*(9/3)-4)*3/2
```

What answer do you expect? Keep this in mind: *Rule:* The computer looks first at the innermost parentheses. The $9/3$ calculation is *nested* within the outer

parentheses. *Nested* is a term often used in computer jargon. It means “one placed within another.”

These are the steps the computer took:

```
( 7 * ( 9 / 3 ) - 4 ) * 3 / 2
then ( 7 * 3 - 4 ) * 3 / 2
then ( 21 - 4 ) * 3 / 2
then 17 * 3 / 2
and 51 / 2
and finally 25.5
```

See how it started from the innermost parentheses and worked outward, shedding each successive pair of parentheses! You could have written:

```
PRINT ((7*(9/3))-4)*(3/2)
```

with the same results. Note that $9/3$ is nested within three pairs of parentheses. There is no limit to the number of parentheses pairs you can use.

Practice with parentheses and the four mathematical relations until you feel you understand how your Timex operates with numbers.

Another math function you are sure to come across is the *power* symbol (** on the H key):

```
2*2 or 4*4
can be written as
2**2 meaning 2 squared
4**2 means 4 squared
5*5*5*5*5*5
or 5 times itself 6 times can be written
5**6
```

Try this:

```
PRINT 5*5*5*5*5*5
```

Then try

```
PRINT 5**6
```

What did you get?

Hint: You cannot type ** using two * on the B key. That is a syntax error!

In a long arithmetic problem the ****** operation takes precedence over the four (+, −, *, /) operations. They are still computed in the order described, but when the T/S 10000 sees a number taken to a power, it does the power first. For example, in

1	*	2	*	3	**	2	+	4
↑		↑		↑		↑		
1		3		2		4		

the computations operate in the order indicated by the arrows. What answer did you get? I hope it was 22.

You should have preceded the problem with the command **PRINT**. True, you can be sure your T/S will compute without the command **PRINT**. But how will you ever see the answer if the computer isn't directed to **PRINT** (display) it?

Incidentally you may have found the computation seemed to take a long time because the display did not appear suddenly. Don't worry, we can make the computation go faster—but we shall leave that for another level.

Experiment with the power symbol on your own, combining the other mathematical functions using parentheses.

Mathematical functions such as square root, absolute, sine, exponents, logs, etc. will be reviewed in higher levels as required.

NUMERIC VARIABLES

So far, we have learned about string variables—which can only be named A\$ through Z\$, although the strings themselves can be any length. Strings are always in quotes, except in the final printout. Later we shall learn some of the things we can do with strings.

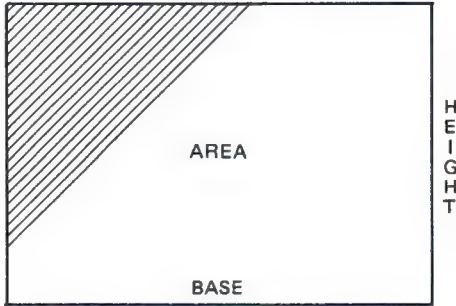
What about variable names for real numbers, as in algebra? You can assign a numerical value to essentially any name or word not in quotes. (You cannot, however, use A\$ through Z\$.) This is an unusual feature in Timex BASIC. Most BASICS limit the numeric variables to a two-letter word. You do, however, have one restriction—the first letter cannot be a number. Otherwise you can mix any combination of numbers and letters (except symbols such as the hyphen, question mark, and inverse characters). This means that if a computation involves “interest,” for example, you could represent it with the symbol “I” or “I3” or, if you wish, “INTEREST”! Using an entire word as a variable might be easier for you while learning to program.

Why do you think most professionals would rather use the letter “I” in the above example? Because they hate to type? Well maybe! But mainly, it takes longer to key in and consumes more computer memory.

LET Statements

To assign a number to a variable, we use the **LET** statement in a program. As a command by itself it is useless. (You will find **LET** on the **L** key as a key-word.)

To illustrate, let's write a simple formula—the area of a rectangle (area equals base times height.)



We would type it thus:

```
20 LET AREA = BASE * HEIGHT
```

The computer will accept this statement but cannot do anything with it until the variables **BASE** and **HEIGHT** are defined. So add the following:

```
10 LET BASE = 5
```

```
15 LET HEIGHT = 4
```

Now we need a statement to print the result:

```
25 PRINT AREA (then RUN of course)
```

The computer will substitute the values of **BASE** and **HEIGHT** in the formula and call it **AREA** and print the value of **AREA** as 20.

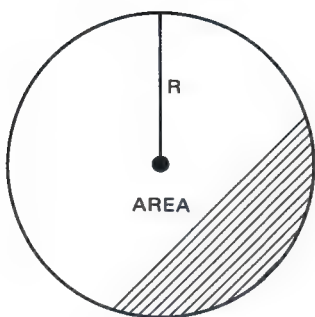
You could be more sophisticated by now and have it display "AREA IS 20" using this statement:

```
PRINT "AREA IS #"; AREA
```

No, I did not forget a line number since the value of the variable **AREA** is still in memory! Note that it is necessary to define **BASE** and **HEIGHT** in the program *before* the computer is asked to define **AREA**.

Let's try for the area of a circle with radius 4 given the formula $AREA = \pi R^2$

```
10 LET AREA = PI * R ** 2 (PI is a function on the M key)
```



```
5 LET R = 4
15 PRINT AREA
```

(Note that even if these lines are put into the computer out of sequence, as here, they will be displayed in sequence.)

To see what value of **PI** is stored in the computer type:

```
PRINT PI
```

(**PI** is represented by the Greek symbol π on the **M** key. **PI** is not the same as the letters **P** and **I** typed together.)

A note of caution is worthwhile here: In a **LET** statement the “equals” sign doesn’t really mean “equals,” as in a formula. **LET AREA = BASE * HEIGHT** really tells the computer to define what is on the left side of the “equals” sign by what is indicated on the right side. That is why a **LET** statement like

```
LET J = J + 1
```

is possible. It says, “Let **J** now be what **J** was before plus one.”

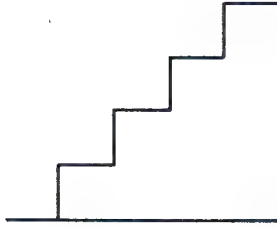
Type in this program:

```
10 LET J = 0
20 LET J=J+1
30 PRINT J
40 LET J=J + 1
50 PRINT J
60 LET J=J + 1
70 PRINT J
```

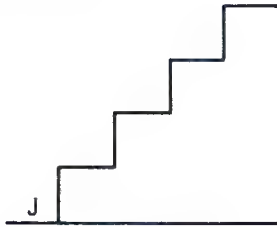
Here’s what it did:

```
10: J = 0
20: J = 0 + 1 = 1
40: J = 1 + 1 = 2
60: J = 2 + 1 = 3
```

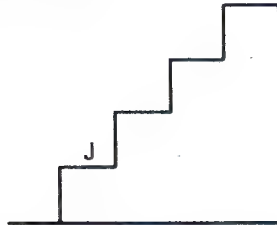

Let's imagine a set of stairs:



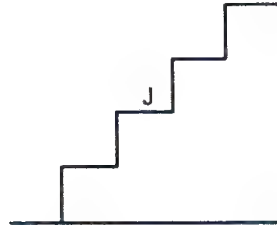
You are J, and at the bottom J is \emptyset :



You take one step. You are now on step 1. J is 1:



You take another step. You are now on step 2. J is 2:



And so on! **LET** means “allow.” You allow J to take on another value. What happens *inside* the computer with a **LET** statement? Your T/S sets aside a spot, calls it J, and puts a \emptyset in it. When you typed **LET J = J + 1** it added 1 to the value of J (in this case, \emptyset) and put the new value in the same spot where the \emptyset was. It is still called J.

Try this more practical program, which prints out the expected population of the United States for every five years beginning with 1980. It is based upon the population increasing 20 percent every five years. So we first need starting points:

40 Creative Programs Using Loops and Decisions

```
10 LET YEAR = 1980
20 LET POPULATION = 220
```

We want to print a heading:

```
30 PRINT "YEAR", "MILLIONS OF PEOPLE"
```

We need to print the year and population:

```
40 PRINT YEAR, POPULATION
```

We want to look at next five years, so we write:

```
50 LET YEAR = YEAR + 5
```

But for that, the population becomes:

```
55 LET POPULATION = POPULATION * 1.2
```

By now you can see why programmers don't use whole words as variables—it is too much repeated typing.

Now we want to print this new value:

```
60 PRINT YEAR, POPULATION
```

Now we only have values for 1980 and 1985. Do we have to keep repeating lines 50, 55, 60? No! We don't—the computer does. That is what computers are for!

Here's how. First rewrite the program in a simpler fashion:

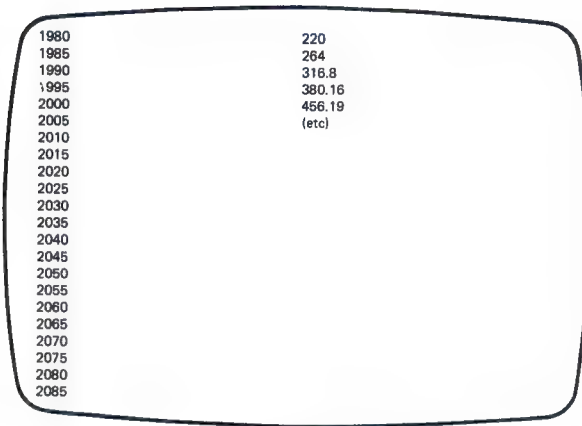
```
10 LET Y = 1980
20 LET P = 220
30 PRINT "YEAR", "MILLIONS"
40 PRINT Y, P
50 LET Y = Y + 5
60 LET P = P * 1.2
```

Now to repeat line 40, type:

```
65 GO TO 40    (GO TO is a keyword on the G key)
Depress RUN
```

Result? the T/S 1000 displayed what was asked. Why the error? Because it ran out of screen space. Depress **CONT** (and **ENTER**) to see what happens. Press

FIGURE 2.1



1980	220
1985	264
1990	316.8
1995	380.16
2000	456.19
2005	(etc)
2010	
2015	
2020	
2025	
2030	
2035	
2040	
2045	
2050	
2055	
2060	
2065	
2070	
2075	
2080	
2085	

CONT again and again. If you were using a paper printer it would continue forever (practically speaking, of course).

If you go far enough, the computer will start displaying very large numbers of population (year 2655) and finally stop at year 4245 due to error code 6—arithmetic overflow (a number greater than 1 with 38 zeros!)

Now let us experiment with **GO TO**. Change line 65 to read

GO TO 20 or **GO TO 50**

How does that affect the results? You don't always have to go backward. Try inserting:

55 GO TO 30

or

55 GO TO 20

What happened?

Of course you don't use this command indiscriminately. You use it in a program when it makes sense. Remember when you used it to replace **RUN** when you didn't want to erase your variables?

When you typed in line 65 you created a *loop* causing the computer to follow a path repetitively—i.e., from line 65 to line 40 to line 50 to line 60 to line 65 and back to line 40 and so on.

LOOPS

Now it becomes obvious you need some way to stop a program within such a loop, for the computer could run forever! There are three such practical techniques: the use of a counter, a signal input, and a special looping statement. We shall develop each in detail.

The Counter

```
1 LET K = 1
2 LET K = K + 1
3 IF K = M THEN [do something]
```

These statements are essentially the counter. Statement 1 is called the initializer—it starts the count. Line 2 is really the counter—adding a unit each time you perform a loop. Line 3 is what stops the loop. It is an **IF** statement. If the value of the counter attains a certain level, the **IF** statement recognizes it and pulls you out of the loop.

In Timex BASIC **IF** must be followed by **THEN**. The key **THEN** automatically triggers the **⏏** cursor and looks for a keyword. This means you can direct it to go somewhere or do something such as **LIST**, **PRINT**, or **LET**.

Now let's see how we can use the counter in the previous population program. Use **NEW** and retype the program. Then assign the following line numbers to the counter and **ENTER**:

```
5 LET K = 1
63 LET K = K + 1
64 IF K = 10 THEN STOP
```

Now **RUN** the program. It will list the population for 10 periods and then stop without error. If you had something special to do, you could write:

```
64 IF K = 10 THEN GO TO 70
```

and line 70 could even be another program.

Supposing you want to interject a line space after 5 time periods. Add

```
62 IF K = 5 THEN PRINT
```

This doesn't take it out of the loop, but says to skip a line before proceeding with line 63. The **IF** statement is a valuable concept in BASIC and will be reviewed in depth as we continue.

Signal Input


A less popular but direct approach is simply to address the input. In the previous example you could simply add this line:

```
62 IF Y = 2020 THEN STOP
```

This is useful if you know where in the information listing you want to stop. This isn't always known.

Look at this sample:

```
110 INPUT X
120 INPUT Z
130 PRINT X + Z
140 GO TO 110
```

RUN it. Use **GO TO 110** so you don't interfere with other programs. This one will take any two numbers and print their total. But it will go forever, as long as you enter digits when the  cursor appears.

You now have two problems: how to fix the program so you can get out of the loop at will and how to get your program back.

There are four ways to get a program back if you are "stuck" in a loop (i.e., the computer won't stop computing).

1. Depress the **BREAK** key (**SPACE** key). This usually works if the computer is computing and not displaying. In the above program it doesn't work.
2. Do something wrong. In this case your T/S is asking for a number since there are no quotes, so type in a letter—you get an error code and can now **LIST** the program. This works on the program above.
3. Fill the screen by continuously responding until you get an error code. (It wasn't necessary in this example).
4. Pull the plug. Sorry, if you had to do this you were wiped out! (Your program was erased when the machine was turned off.)

LIST and add:

```
115 IF X = -99.99 THEN STOP
```

The number -99.99 was chosen because it is highly unlikely to come up.

Now **RUN** this program:

```
110 INPUT X$
120 INPUT Y$
130 PRINT X$ + Y$
140 GO TO 110
```

(Line 130 "adds" two string variables.)

Respond to the program by typing in the two words **BOOK** (for X\$) and **KEEPER** (for Y\$). You probably didn't think you could add *words*, did you? It's really not adding, but "cocatenation," meaning "connecting in a chainlike fashion." Hence "BOOK" added to "KEEPER" results in "BOOKKEEPER".

In any case, try to get out of this loop! The **BREAK** key results in a **SPACE**. There is nothing you can insert in quotes illegally, therefore procedure 2 is useless. You could keep responding to **INPUT** by keying in words until it runs down to display line 21, getting an error code 5.

FOR . . . NEXT Loop

Finally, this is the most versatile and popular means of controlling loops. This technique includes in only two lines an initializer, a counter, and an efficient stopper. It looks like this:

```
1 FOR I = 1 TO 10 STEP 2
20 NEXT I
```

This is what it is saying:

Start a counter at $I = 1$.

Count up to 10 in increments (steps) of 2.

When 10 is reached, leave the loop.

Syntax: A **FOR . . . NEXT** loop always begins with a **FOR . . . TO . . . STEP** statement and ends with a **NEXT** statement. Everything in between is part of the loop. Note that no punctuation is used in the **FOR . . . TO** statement.

Depress **NEW** and **ENTER**.

To see what happens with the loop **RUN** the following:

```
10 FOR I=1 TO 10 STEP 2
15 PRINT I
20 NEXT I
```

Hint:

FOR is on the **F** key

TO is a **SHIFT 4**

STEP is a **SHIFT E**

NEXT is on the **N** key

What result did you expect? You should get the numbers 1, 3, 5, 7, and 9. It will not go past 10. Notice it went in steps of 2.

Practice with this program. Go back and **RUN** it several times, each time changing one of the numbers in line 10 to see what it does.

Hints: Only a single letter can be used for the counter. There is no limit to the number of times you loop. The step increment number can be any number less than the number of loops and can also be noninteger, negative, a variable, or not be there at all! (*Note:* “integers” are whole numbers, without decimal points.)

RUN the above program with **STEP 1**, then without the **STEP** word at all. Was it the same?

Try this program:

```
10 FOR I = 1.2 TO 34.4 STEP 1.1
15 PRINT I
20 NEXT I
```

Now change line 10 to read:

```
10 FOR I = 1.2 TO 34.4
```

If you do not indicate **STEP**, **STEP 1** is assumed. How do we apply this to our original population program? Delete lines 10, 50, 65, 110, 115, 120, 130, and 140. Add lines 35 and 70 or retype it as shown:

```
20 LET P = 220
30 PRINT "YEAR", "MILLIONS"
35 FOR Y = 1980 TO 2020 STEP 5
40 PRINT Y, P
50 LET P = P*1.2
70 NEXT Y
```

Now **RUN** it and note how more efficient this program is.

Now let's try some picture loops.

FUN LOOPS

Let's see what we can do with looping. **RUN** this one:

```
10 FOR I = 1 TO 5
20 PRINT "*****"
30 NEXT I
```

Let's get fancy with a "nested" loop:

```

10 FOR W = 2 TO 8 STEP 2
15 PRINT "*" * 4      (Remember: # means SPACE)
20 FOR X = 18 TO 20
25 PRINT "*" * 4
30 NEXT X
35 NEXT W

```

You can nest as many loops as you want—but make sure you don't jump out of a loop by putting the **NEXT** statements in the wrong nest! That is, in the above program if you had 30 **NEXT W** and 35 **NEXT X** you would incur a different result or possibly an error.

Now try this one using a variable:

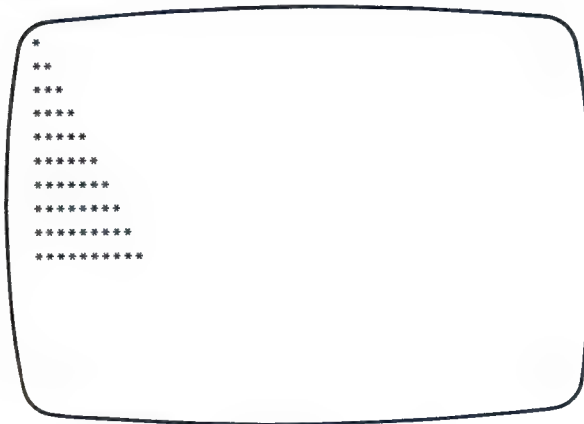
```

10 FOR S = 1 TO 10
20 FOR T = 1 TO S
30 PRINT CHR$( 23);    (CHR$ is a function on the U key)
40 NEXT T
50 PRINT
60 NEXT S

```

SURPRISED?

FIGURE 2.2



Delete line 50 and see what you get. Can you follow the computer's logic?
The inner loop:

```

LET S = 1
FOR T = 1 TO S

```



```
PRINT CHR$ 23;
NEXT T
```

Initially it went through the inner loop only once because *S* is 1. When completed, it printed one asterisk (*). Then it set *S* = 2 and printed a second line by going through the inner loop *S* times. So it printed two asterisks. And so on until 10 lines were printed, each with one additional asterisk.

The semicolon after **CHR\$ 23** allows it to print consecutive asterisks. Line 50 instructs the computer to start a new line.

See how powerful the loop can be!

OK—so now what was that **CHR\$** character all about? Well, remember the interest stimulator in Level 1—where it printed **CHR\$ N**, where *N* was a variable from 1 to 255?

Well, every symbol, word, letter, and number on the keyboard has a place in the computer. A subsequent level will discuss memory, CPU, chips, etc., but for now note that there are 255 places within the computer. They are coded as listed in your manual. You can therefore call for any character you want using the **CHR\$** function. Keep in mind that the dollar sign means that even if you call for a number, it appears as a string alphanumeric, not as a number. For example, 8 is not a number representing a quantity but rather a symbol consisting of two small “ohs”.

In the last fun program which started with a variable *S*, change line 30 to read:

```
30 PRINT CHR$ 6
```

and **RUN** it. You are beginning to see how a checkerboard could be made!

You'll see this same symbol on the **T** key. It can be obtained directly by entering **GRAPHICS** mode (**SHIFT 9**). Note the “G” cursor. Once you have the **C** cursor it remains fixed until you depress **SHIFT 9** again. Push various keys in this mode—you will see everything printed white on black, otherwise known as *inverse video*. Keys with symbols in the bottom right corner will print the symbols in *graphics mode* with **SHIFT**. If there is no graphic symbol, your T/S will print the key's upper right corner symbol in inverse video (when in **SHIFT**).

So you see you can type line 30 as 30 **PRINT** “[any character]”; with whatever graphic symbol you want.

Go ahead—experiment! Try 30 **PRINT CHR\$ T** in the above program.

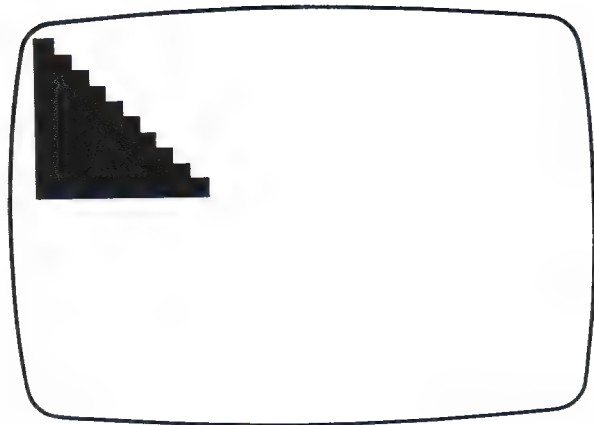
Want to make some stairs? Replace line 30 with

```
30 PRINT CHR$ 128
```

CHR\$ 128 is the inverse of space. How did we know the code number? Your Timex knows all—just ask it by typing:

```
PRINT CODE “[any character]” followed by ENTER
```

FIGURE 2.3



Insert in between the quote marks any character for which you want the code number, since the **CODE** function (I key) only works on a string. The inverse space symbol (**CHR\$ 128**) is also obtained by getting into **G** mode and depressing the **SPACE** key, then getting back into the **L** mode.

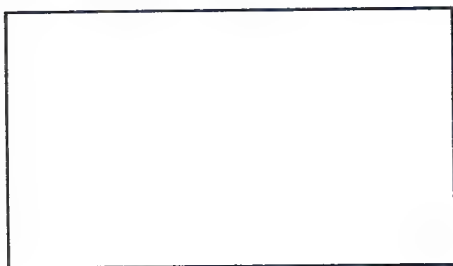
HOW TO WRITE A PROGRAM

Now you know some BASIC vocabulary and some rules, and you've seen some programs. You can even interpret written programs. But how do you write them from scratch?

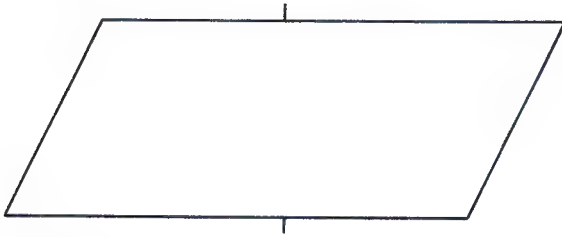
Well, you must think like a computer. Do only and exactly what you are told. Studying other programs helps. But it is not hopeless—there does exist a basic tool to use, common with all computer languages. It is called the *flowchart*.

Flowcharting is a technique of using diagrams to illustrate the sequence of computer operations in a program. The technique requires five basic chart symbols:

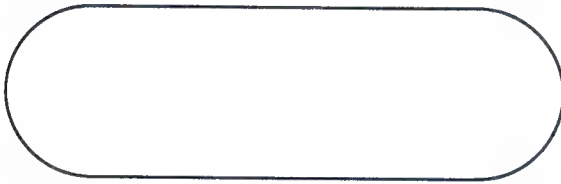
1. The rectangle is used to represent processing, such as LET statements.



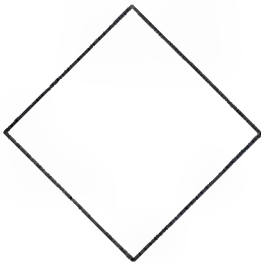
2. The parallelogram is used for input and output statements such as the INPUT and PRINT statements.



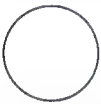
3. The elongated oval is used for termination—that is, the beginning and ending.



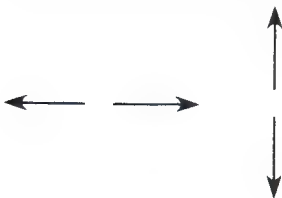
4. The diamond is used for decisions—such as the IF statement. Choices branch out from here.



5. The circle is used as a connector for convenience only.



6. Arrows are used to show the direction of flow.



As a simple application, consider the following problem: Johnnie is told to go to the A&P and pick up eggs, Wonder Bread, and milk. However, if the A&P does not have Wonder Bread he is to go to the 7-11 and find it there; and if the A&P also does not have milk or eggs, he is to go to Shop-Rite and buy *everything* there. Once everything is purchased he is to return home and do the same for his neighbor. His steps using a flowchart are depicted in Figure 2.4.

If you follow the lines from start to stop, Johnnie will have made two trips and purchased all he needed at one or two stores.

Now let's look at a program already studied and see how the flowchart (Fig. 2.5) would have been written.

```
20 LET P = 220
30 PRINT "YEAR", "MILLIONS"
35 FOR Y=1980 TO 2020 STEP 5
40 PRINT Y,P
60 LET P=P*1.2
70 NEXT Y
```

As promised, you should now know how to use (1) math symbols; (2) numeric variables; (3) the **LET** statement; (4) the **IF . . . THEN** statement; (5) the **FOR . . . NEXT** loop; and (6) flowcharts. Therefore, you are ready to handle the Sample Program involving wages presented earlier in this level:

1. Identify the input data and assign names for programming. In this case we have:

NAME	N\$
HOURLY RATE	R
HOURS WORKED	H

2. Define the output wanted and assign programming names. In this case we want:

a. Headings—including their literals and data names:

Employees' names	NAME	N\$
Hourly rates	RATE	R
Hours worked	HOURS	H
Gross pay	GROSS	G

b. Output results:

Same as heading N\$, R, H, G

3. Identify the processes to get your outputs:

Gross pay = hourly rate times hours worked or
 $G = H * R$

FIGURE 2.4

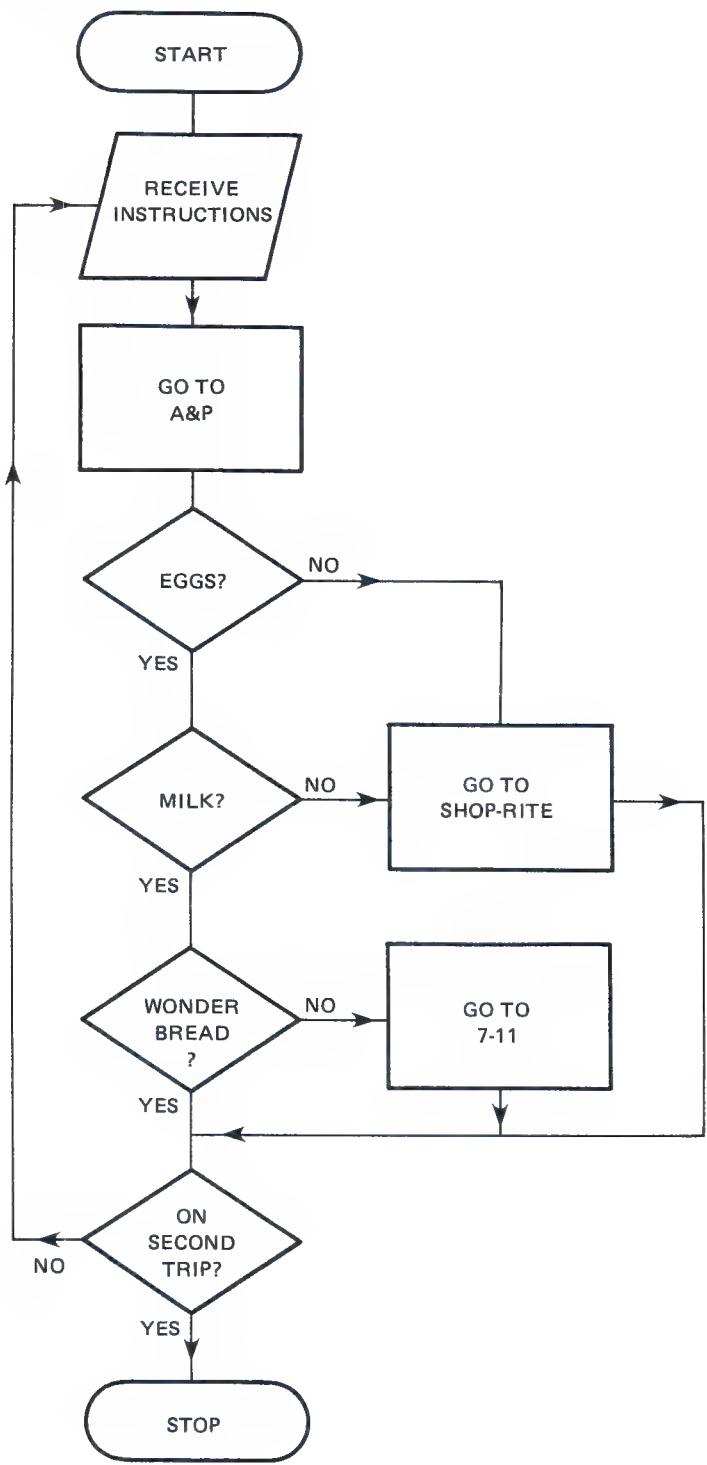
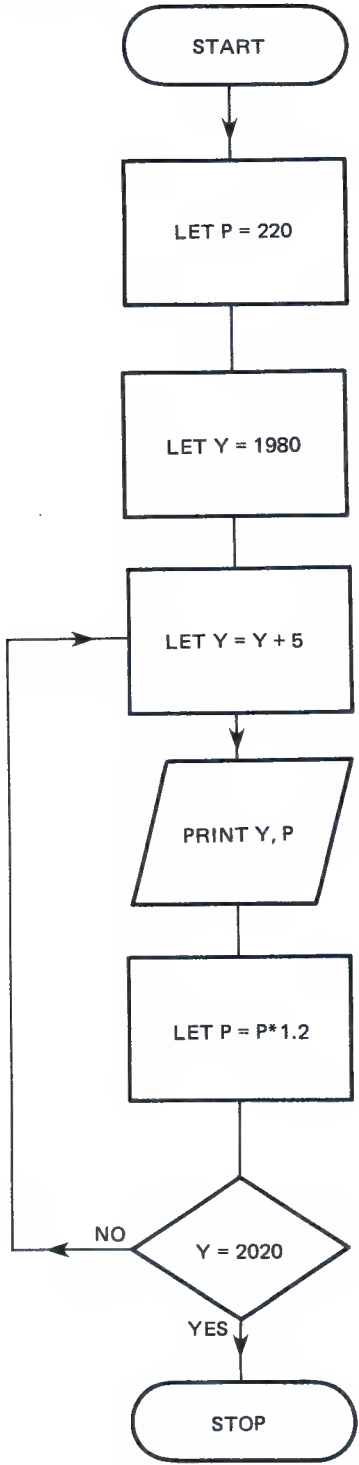


FIGURE 2.5



4. Note how much data we have and in what form. Data must be input in the Timex one by one. We have three employees, or three sets of data.
5. Create a flowchart as in Figure 2.6.

With the flowchart we shall now write the rough program:

```

Ø5 REM WE PRINT THE HEADING FIRST AND LIST ALL
  INITIALIZERS IF ANY
Ø7 LET K=1
1Ø PRINT "NAME", "HOURS", "RATE", "GROSS"
15 REM NOW LIST THE INPUTS
2Ø INPUT N$
3Ø INPUT H
4Ø INPUT R
45 REM NOW DO THE PROCESS
5Ø LET G = H * R
55 REM NOW PRINT THE RESULTS
6Ø PRINT N$, H, R, G
65 REM PUT IN A CHECK TO SEE IF WE HAD THREE EMPLOYEES
7Ø IF E = 3 THEN GO TO 11Ø
75 REM PUT IN COUNTER AND PUT IN INITIALIZER AT START
8Ø LET K = K + 1
85 REM PUT IN LOOP
9Ø GO TO 2Ø
95 REM GIVE SOMEPLACE FOR LINE 7Ø
11Ø STOP

```

Now **RUN** this program and see what is wrong with it.

You will note the following:

1. Your **PRINT** statements should use **TAB** and **AT** functions to better arrange the heading and results in accordance with the special characteristics of the Timex.
2. You can substitute a **FOR-NEXT** loop in place of a counter. Cancel line 7 and add line 17:

```
17 FOR E = 1 TO 3
```

Delete lines 7Ø, 8Ø, 9Ø, and 11Ø and type line 7Ø as:

```
7Ø NEXT E
```

FIGURE 2.6

1. Begin with a starting point

2. Print the heading

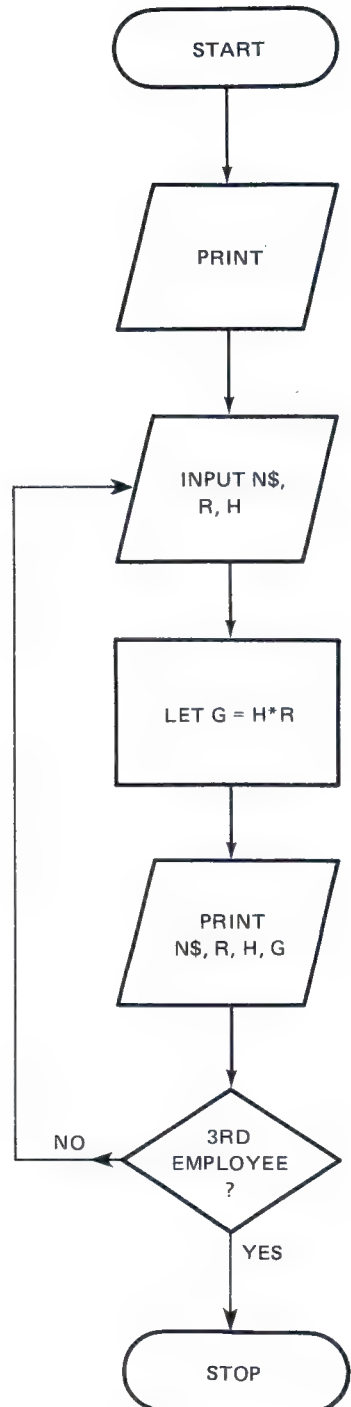
3. Input the data

4. Define the process

5. Display the output

6. Make a decision: Have we included all employees?

7. If no—loop back and do another.
If yes—end the program.



3. You are missing display words requesting input information. It is fine if you remember what the program requires, but what about someone else? So you add print statements before each input:

```
19 PRINT "NAME?"
29 PRINT "HOURS?"
39 PRINT "RATE?"
```

Your program will probably look like this:

```
10 PRINT "NAME"; TAB 14; "HOURS"; TAB 20; "RATE"; TAB 26;
   "GROSS"
17 FOR E = 1 TO 3
19 PRINT "NAME?"
20 INPUT N$
29 PRINT "HOURS?"
30 INPUT H
39 PRINT "RATE?"
40 INPUT R
50 LET G = H * R
60 PRINT N$; TAB 15; H; TAB 20; R; TAB 26; G
70 NEXT E
```

RUN this and you will find that you need some fine-tuning. Now that we have the cues printed (lines 19, 29, and 39), they interfere on the screen with the final display.

Can we stick in a **CLS** command? No—it will clear the entire display as well.

So we come up with some tricks. Position the display lower by starting the **PRINT** statements with an **AT** function.

Also force the cue questions to stay on one line using the **AT** function, and finally eliminate the cues altogether by printing spaces.

So change these lines to read:

```
10 PRINT AT 4, 1; "NAME"; TAB 14; "HOURS"; TAB 20; "RATE";
   TAB 26; "GROSS"
19 PRINT AT 20, 1; "NAME?"
29 PRINT AT 20, 1; "HOURS?"
39 PRINT AT 20, 1; "RATE?#"
60 PRINT AT 6, 0; N$; TAB 15; H; TAB 20; R; TAB 26; G
```

(Note: These are easily made using the **EDIT** key.) And add line 41:

```
41 PRINT AT 20, 1; "#####"
```

After you **RUN** with these changes you discover you are stuck on line 6. You want all employees' data to display, not just one. So change the **AT** function in line 60 to read:

AT 5 + E, 0

and your efforts will finally have paid off!

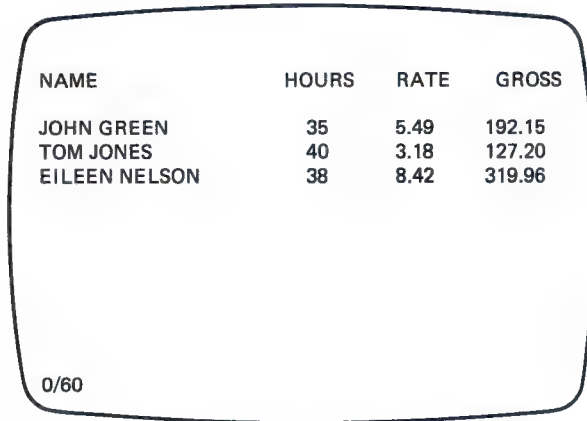
Given the data, your results should be displayed near the screen top:

NAME	HOURS	RATE	GROSS
John Green	35	5.49	192.15
Tom Jones	40	3.18	127.20
Eileen Nelson	38	8.42	319.96

You might want to improve the display further by putting in some dollar signs!

Can you think of other things you might add?

FIGURE 2.7



You can see that writing a program is no easy chore. However, once written it can save hours of work since the computer can perform its function over and over without error or tiring out.

You may not want to write programs, but you can copy written programs and store them on tape for reuse. You can also purchase someone else's tapes with some idea of what you are buying.

SUMMARY REVIEW

Although we have hardly covered all of the keys on your Timex, you have learned enough in two levels to write quite a few programs.

You have added the following terms to your vocabulary: **NEW, CLEAR, CONT, STOP, BREAK, LET, FOR . . . TO . . . STEP . . . NEXT, IF . . . THEN, CHR\$, CODE**

You have learned how to loop and how to stop looping. You have learned how to stop a computer out of control.

You are now familiar with all the cursors, including the **GRAPHICS** cursor, **K**, **L**, **S**, **F**, **G**, and **█**.

You know two ways of displaying graphic symbols.

You understand how programs are written using flowcharting.

You know the difference between string and numeric variables and how they are treated differently.

You know how the computer performs arithmetic, and you are familiar with the four basic arithmetic operations, as well as powers, pi, and the use of parentheses.

You also now understand how Level One stimulators worked. Level Three will give you a break until we dive into more programming in Level Four.

INTEREST STIMULATORS

Now for more advanced interest stimulators:

1. Here's a simple one using something you already learned and something you haven't:

```
10 PRINT CHR$(INT(RND*11) + 128);
20 GO TO 10
```

(Note: **INT** is on **R** key; **RND** is on **T** key.) The error you get only means you ran out of space.

2. This one also involves functions and commands yet not covered:

```
10 FOR J=1 TO 4
15 SLOW
20 IF J/2=INT(J/2) THEN FAST
30 LET DOTS=48
40 GOSUB 100
50 LET DOTS=40
60 GOSUB 100
70 LET DOTS=55
80 GOSUB 100
90 NEXT J
100 FOR L=1 TO DOTS
```

```

110 PLOT 32+(L*22/DOTS)*COS (L/DOTS*PI*100), 22+(L*
    22/DOTS)* SIN(L/DOTS*PI*100)
120 NEXT L
130 PAUSE 100
140 POKE 16437,255
150 CLS
160 RETURN

```

You will fully understand these after you attain Levels Four and Five.

EXERCISES

- 2.1 Given a triangle with base 50 and height 10, write a program to find the area of any triangle and specifically this one.
- 2.2 You can nest as many arithmetic problems with parentheses as you need, but you can only have four nested loops. True or false?
- 2.3 You have the results of a program displayed. You cannot see the listing. You want to **LIST** the program and make a **PRINT** change on line 60. You choose which one of the following?
 - a. Depress **LIST** 60, **EDIT**, make change, and **RUN**.
 - b. Type in 59, **ENTER**, depress **EDIT**, make change, and type **GO TO** 60.
 - c. Depress **CLEAR**, then **LIST**, move arrow cursor to line 60, **EDIT** and make change, and type **GO TO** 60.
 - d. Immediately type 60 **PRINT** . . . etc. as corrected. **ENTER** and **RUN**.
- 2.4 Is this program legal?


```

10 FOR 2 = 1.1 TO 10.4 STEP 1.1
20 PRINT Z$
30 NEXT Z

```
- 2.5 What is wrong with the following program?


```

10 FOR X = 1 TO 10
20 FOR Y = 2 TO 4
25 LET Z = X * Y
30 PRINT Z,
40 NEXT X
50 NEXT Y

```
- 2.6 Answer true or false: **STOP** is a keyword.

- 2.7 All words which respond to the **K** cursor are called keywords. They are also commands. True or false?
- 2.8 You are writing a program which has 30 lines. As you type them in, the top lines disappear. Where did they go? And how do you retrieve them? Can you use **CONT** to show the rest of the program?
- 2.9 Which of the following are not acceptable numeric variables? NAME, BACON AND EGGS, A234, B?QX, 3TOM, B.
- 2.10 What results would you expect from these statements?
- PRINT** 6*2**2*1/2+2**3-20
 - PRINT** 2*4+2+1+5/2*4-3*7
 - PRINT** (1*((4*(2*(3+5)+2)/2)-36))
- 2.11 Is the following a legitimate command to determine the area of a circle?

PRINT "AREA OF A CIRCLE IS #"; PI*R2**

- 2.12 Write a brief program to find the volume of a cube with side of S.
- 2.13 Write a program based on the **LET** statement **LET Y = J + 1** and **FOR NEXT LOOP** to print the numbers 1 through 10.
- 2.14 Can you think of another use for a counter besides determining when a loop is done?
- 2.15 Which of these can you add after **THEN** in 10 **IF K = M THEN**?
- GO TO J**
 - PRINT "HELLO"**
 - RUN**
 - STOP**
 - CLS**
 - LET**
- 2.16 Write a program that will print the digits 1 through 9 on a single line. Use the following **PRINT** statement: **PRINT N;"#";** (# means space.)
- 2.17 Write the **LET** statement which will multiply 4 times the sum of A and B and store the results in C.
- 2.18 Convert the following formula into the appropriate BASIC statement:

$$K = \frac{3(A + B + C) - P}{N^2} + 6$$

- 2.19 What is wrong with this statement?

10 **FOR** 10 **TO** 1 **STEP** - 1.33

- 2.20 Can you print any symbol within quotes (literals)?
- 2.21 If you need the code number for a symbol so you can use the **CHR\$** function, how do you get it?
- 2.22 What would the result be of the following?

PRINT CODE "NAME"

- 2.23 Give a reason why you might use inverse video.
- 2.24 Why do we have two types of variables—numeric and string, and not one combined? Can we change from one form to another?
- 2.25 *Problem:* You own a small company selling computers. You have three salesmen. Each earns a commission based upon his sales. It is figured at 10 percent, but if a salesman sells over \$1000, he is encouraged with 15 percent on all. This month you have the following receipts:

SALESMAN	NUMBER	SALES
T. Crivaro	120	45, 2007, 314, 105
P. Silvers	683	15, 31, 9, 18
J. Rothberg	314	29, 91, 54, 50

Display your salesmen's names, numbers, total sales, and commissions. Also show the total commission you had to pay out.

Hints:

- Use a flowchart.
 - Treat each salesman's number as a string variable. Your input statement will ask for salesman, I.D. number, sales 1, S2, S3, S4.
 - You will calculate total sales and commissions and maintain a running total.
 - You will decide which commission rate to use.
 - Do not enter large numbers into the computer with commas—remember commas are instructions. For example, \$15,000.00 should be entered as 15000.00.
- 2.26 Now a problem with no help! You have suddenly inherited a trust estate of \$10,000. It will earn 15¼ percent annually, or monthly interest of 1.3125 percent. You automatically receive \$100.00 a month until your balance is under \$1000.00.

For how many months will you receive a check of \$100.00. And what will the final balance be?

Solve this problem using the flowchart technique, identifying inputs or initial data, computation, decisions, and printouts. Use a **FOR . . . NEXT** loop.

LEVEL THREE

HARDWARE AND MEMORY

COMPUTER TERMINOLOGY

Now that you have some experience with your computer and can appreciate its versatility, you might be wondering what makes it work. If you have been exposed to any computer literature, you have probably become curious about the words or acronyms RAM, ROM, etc. Learning their meanings will help you better understand your Timex and consequently contribute to your programming skills.

Hardware

Your microcomputer consists physically of a case, a printed circuit board, and a keyboard. The printed circuit board includes a modulator, input and output jacks, a voltage regulator and heat sink, resistors, capacitors, and diodes, and four integrated circuit (IC) chips.

The flat, pressure-sensitive keyboard is the input device, and the TV or video display is the output device. A cassette tape or floppy disk system are external storage devices. The modulator is a device which sends computer signals to the television antenna inputs. A printer is also an output device. All the above is considered *hardware*. *Software* is the information documentation and programs used

to instruct the computer. The output on a TV screen is called soft copy. The output on a printer is called hard copy.

The core of your computer lies in the ICs. These are the small black plastic blocks about 1 to 2 centimeters wide and up to 8 centimeters long, with 20 to 40 leads coming out of each for attachment to a socket on the printed circuit board. They look like black caterpillars!

As small as these plastic blocks are, the actual electronic chip is one-tenth as small and embedded in the middle. The chip is essentially a giant electronic logic circuit condensed into a miniaturized "city."

Each of your ICs is specialized. The four ICs in your Timex are called the CPU, ROM, RAM, and SLC chips. The CPU or central processing unit is the heart (or perhaps, more aptly, the brain) of the computer, for it controls what the computer does. It has three main sections: arithmetic operations, control system, and logic elements. It performs calculations, provides for television display, and oversees other chips. It is also known as a Z-80A *microprocessor*.

The second major chip is the 8K ROM chip. This is the translator electronic circuit that enables you to use English or Timex BASIC with it. The chip translates BASIC into machine language which the CPU can process. Your Timex will also accept machine language directly, saving both time and memory. ROM stands for read-only memory. This means it is a fixed logic wiring circuit. Given certain electrical impulses—input—it always follows the same route and gives identical results. ROM chips cannot be changed by the programmer. The "Speak and Spell" modules and Atari's game cartridges you may be familiar with are all ROM chips designed for a specific purpose. The Timex ROM is designated 8K because it takes up over 8000 storage locations called bytes within the computer to store the language. You might say it is a dictionary of 8K bytes!

If you are interested in keeping abreast of new hardware (and software) for your computer, join a users' group or subscribe to one of the publications devoted solely to the Sinclair machines. Suggested magazines are:

SYNTAX

Box 457

Harvard, Maine 01451

SYNC

39 E. Hanover Avenue

Morris Plains, NJ 07950

The third important chip is the RAM chip. RAM means random-access memory. This chip is not prewired but is essentially a series of miniature on-off switches called *bits*. The standard memory chip (#2114) is a $1K \times 4$ chip, meaning it has 4096 switches arranged in 4 rows of 1024 bits each ($K = 1024$). The $1K \times 4$ chip then actually has 4096 bits. Computer memory is classified in bytes. Your computer and all others based on the Z-80 processor require 8 bits per byte. Hence, for 1K RAM you need two $1K \times 4$ chips, which equals $1K \times 8$ bits or 1K bytes.

What can be done with all these bits and bytes? In general, each letter or character in a program requires one byte. Some special BASIC instructions may require more. It is very easy to use 1000 bytes. In fact, many commercial computers need 100,000 bytes and more.

Why does a byte need 8 bits? Because information is stored in machine language using binary code. The binary number system is based upon 2 rather than 10.

The binary system only recognizes two characters, 0 and 1, which electronically can be interpreted as on or off, true or false, open or closed. Without delving any further into binary code, be aware that a "1" is 00000001 and a "2" is 00000010. The largest information-code number possible then is 11111111, which equals 255 in our decimal system. This is why the codes are limited to 255 characters. Further explanation of computer space and how it is addressed will be reserved for a subsequent level. For now, it is of value to know how much memory you consume when you write programs in BASIC. Efficient programs work faster and require less valuable memory, leaving more space for variables.

The fourth chip, the SCL, is the Sinclair Computer Logic chip, named after the designer, Clive Sinclair. It is the specially designed circuit that ties all the chips together.

Byte Consumption

Here is how your bytes are eaten:

1. Any numerical constant consumes 6 bytes for storing the number plus 1 byte for each character in the number for display. Hence "24" requires 2 bytes (1 per character) plus 6 for storing the value of the number.
2. Any other character (letter or symbol) consumes 1 byte.
3. Typed spaces in a **REM** statement or in a string quote consume 1 byte each.
4. Commands and single-keystroke words consume 1 byte each (an advantage).
5. Line numbers—whether 01 or 9999—consume a total of 5 bytes each. This includes 2 for the line number itself, 1 for entering the line, and 2 to store the line.

Example: The statement 10 **LET** P=1 uses 5 bytes for the line number, 1 for the command **LET**, 1 for the letter P, 1 for the symbol "=", and 7 for the number "1"—or a total of 15 bytes. However, 10 **LET** P=PI/PI uses 4 less bytes because **PI** only consumes 1 byte each time it is used and the "/" mark also uses only 1.

Comment: With 2K RAM in the Timex, you have 2×1024 or 2048 bytes. You can conserve bytes by minimizing the use of real numbers and using variables where possible, as in the above example, where the number "1" was replaced with **PI/PI**.

Although 1K RAM is in the computer, it is not all available to the programmer. The ROM chip consumes 125 bytes to record internal information. Several hundred bytes are consumed in necessary ways depending upon the program—including the display, work space, and calculator stack. Hence, for practical purposes only 600 bytes or so are available in the 1K machine. Additional chips, however, will be all yours. Hence when Timex offered 2K in their ZX81 machine they didn't offer twice the available memory but really over three times! This is considerable and can handle most programs. The 16K RAM makes your computer a real working tool.

For the programs in this book 600 bytes is sufficient. But if you are interested in storing large amounts of data, the 16K RAM Pack is indispensable. Also, a 64K Memopack is now available.

Assuming you'll use your first 1K RAM for the program, the second K is sufficient for storing about 200 sets of data (where a data item requires 6 bytes). Now 16K is equivalent to 3000 items, and the full 64K allows 9000 sets of data!

Of course, this only involves storage of data within the computer for it to act upon. You actually can store indefinite amounts of data in external storage and retrieve pieces as needed. External storage would include tape cassette or floppy disk.

In the back of your unit you'll find a 44-edge connector. This includes direct connections to important input and output lines of the CPU and the built-in electronic timer as well as electrical ground and regulated power. It is the expansion port for adding memory, printer, modem, or other interfaces. (A *modem* is a device which enables your computer to talk by telephone and tie into a larger computer.)

In summary, you have a special and unique computer. Its characteristics include low cost, small size, sealed waterproof pressure-sensitive keyboard, single-keystroke capability, rejection of syntactically incorrect instructions, automatic error-detection feature, flexible editing, automatic spacing of single words, adjustable dynamic memory, extensive graphics, floating decimal, and 24×32 display.

SOFTWARE

At present software for your Timex comes in two forms—on paper and on cassette type. Programs are written out in detail in many books dedicated to your computer. Books listing programs for other computers such as the TRS-80 or Apple will be of value to you when you complete this book. Then you will know how to convert from other BASICs to Timex BASIC and vice versa.

The same is not true for cassettes. Tape stores electronic signals based upon the program bytes. Hence tapes are *not* interchangeable among different BASICs.

As you are exposed to opportunities to purchase prerecorded programs you need to know what will work on your Timex. If you find software for the following units, you can purchase them knowing they will work on your unit:

Timex/Sinclair 1000
 ZX81
 ZX80—8K ROM

However, make sure the memory requirement (RAM) is less than what you have on your machine. If you have no add-on memory, you have 2K built into the T/S 1000 or 1K in the ZX81. For example, a program designed for the T/S 1000 will not work on the ZX81 (with only 1K) unless you add the 16K package. A 16K program will work with either unit. Programs designed for 16K do not necessarily consume 16K but more than likely need a little more than 2K; they are rated 16K, however, because that is the smallest package available.

A word of caution for ZX80 owners who have upgraded their units with an 8K ROM chip: Many programs designed for the T/S 1000 or ZX-81 include the **SLOW** command and therefore will not work on your unit properly.

If you are beginning to feel “personal” about your personal computer you might be interested in knowing how it was developed. You will find a short history in the Epilogue.

Should you find the 64K packages appealing, be aware your Timex cannot deal with anything beyond 16K for programming. At best you can program your computer to accept an additional 32K, but it can only be used for storing variables. The 64K is misleading; it really only adds 56K: 16K programmable, 32K for variables, and 8K for machine language. The other 8K is already in the computer as ROM. Hence in truth you have 64K addressable locations, although not all will accept BASIC commands.

When additional memory packs are attached, your T/S automatically reviews the memory terminals in the CPU and assesses how much is available. That is why it takes a few seconds for the **K** cursor to appear when the 16K is attached. It takes longer to look at 16K than at 2K!

You can “ask” the computer how many bytes are attached or even how many bytes your program consumed. This procedure will be taught in a later level.

To determine memory manually:

1. Count all figures, letters, symbols, and single words as one byte each.
2. For every real number add 6 bytes.
3. Add 5 bytes for every line which begins with a line number.
4. Add all of the above.

To determine memory left:

1. Determine memory capacity 1K, 2K, 16K, etc., remembering in computer terminology K represents 1024 bytes.
2. Subtract 125 bytes consumed by the interpreter.
3. Subtract your calculation above.
4. This is what is left.

INTEREST STIMULATOR

```

10 PAUSE 40000
20 POKE 16437,255
30 IF INKEY$ = CHR$28 THEN GO TO 65
40 IF INKEY$ = CHR$121 THEN GO TO 68
50 PRINT INKEY$;
60 GO TO 10
65 PRINT "#";
70 GO TO 10
75 PRINT
80 GO TO 10

```

This program allows you to type like a typewriter. TRY it! If you type `Ø` you will get a space; **FUNCTION** will start a new line.

EXERCISES

- 3.1 Peripheral hardware for these units are interchangeable: Sinclair ZX81, Sinclair ZX80 with 8K ROM, Micro Ace with 8K ROM, and Timex 10000. True or False?
- 3.2 When buying software for any of these units it will work on the Timex and vice versa. True or False?
- 3.3 You had a 16K RAM pack added to your unit. Each time you turn on your unit you must key in **REM MEMORY 16K**. True or False?
- 3.4 Look at the answers to problem 26 of Chapter 2. Not counting the **REM** statements, how much memory was consumed by each of the programs. Which is most efficient?
- 3.5 Based on results of question four, would you be concerned if you had the ZX81 with only 1K? How much memory do you have left? What if you have the Timex with 2K or 16K?

LEVEL FOUR

CURVES, SHAPES, AND ARRAYS

If you have reached this level, you have already learned a lot. Programming in BASIC is no longer Greek. There are, however, still many Timex capabilities that are a mystery to you, although you may have tried some functions without really understanding them.

This level will provide you with the skills you need for graphics, charts, and gaming.

Your new vocabulary will include all the trigonometric functions plus **LN**, **EXP**, **SQR**, **ABS**, **SGN**, the graphic symbols, **PLOT** and **UNPLOT**, and special keys **RAND**, **RND**, and **INT**. You will understand the *pixel* and become familiar with points on the screen more detailed than lines and columns.

RANDOMIZING

One of the most popular features of the computer is its capability to produce numbers at random. With such a feature you can choose lottery numbers, create an electronic dice game, and play games with “unknowns.”

There are several techniques for generating a random number in a computer. It generally depends upon the language used and the basic computer hardware

design. The Timex 1000 has a Random function—**RND** on the **T** key. It is not truly random, although it appears to create a new number each time. Statisticians have come up with random series of numbers. This fixed series is tied to the **RND** function. It follows a fixed series of 65,536 numbers. This means each time you turn on the computer and ask for **RND** you will always receive the same series of numbers. Of course, only if you memorize the whole random set can you figure it out. This feature is called pseudo-random. **RND** has no “argument”; i.e., there is no number or variable attached—it stands by itself. If you type **PRINT RND**, it will result in a number between 0 and 1 including 0 but not including 1. You can cause the **RND** function to start with any number in the random series more closely resembling a true random by using the command **RAND A**, where **A** is any number between 1 and 65535. **RAND**, short for “randomize,” is a command on the **T** key. **RAND** is *not* the same as **RND**. **RAND** is a command in the **K** mode and **RND** is a function in the **F** mode. Whereas **RND** obtains a number from a fixed series, **RAND** moves that series around!

For a more realistic randomness you can use the command **RAND 0** or just **RAND**. This works differently than **RAND A**, because it is now tied to the length of time the TV has been on—a true variable! In any case, the command is given somewhere in the program prior to the **RND** function.

Now how do you get real numbers greater than 1 for most practical uses—such as a 6-sided die? We multiply the **RND** function by 6 like this:

```
LET Y = RND * 6
```

However, your results would be numbers between 0 and 5.999. You want to produce an integer from 1 to 6, so you use the **INT** function. This function, however, rounds *down* any number. So 1.2 or 1.99 would become 1. So to create a die of 1 to 6 you use the **INT** function on the **RND** function, then add 1 to eliminate 0 and insure 5.9 becomes 6.9 or subsequently the integer 6.

So **LET Y = INT (RND * 6) + 1** is your electronic die. (**INT** is on the **R** key.)

If you have numbers to round off using the **INT** function, you simply add 0.5 first so that 1.49 rounds off to the integer 1, and 1.51 becomes 2.01 rounded to the integer 2! (Note: enter a decimal less than one as 0.5 or .5. The zero is used for the reader—the computer doesn’t care if it is there or not.) Try it!

```
PRINT INT (1.49 + .5)
```

Then try:

```
PRINT INT (1.51 + .5)
```

How do you round off numbers? Always add 0.5 and apply the **INT**eger function! How about rounding off a number like \$10.638 so it makes sense? Try:

```
PRINT INT (10.638*100 + 0.5)/100
```

Do you see why it was necessary to multiply by 100? No? Then try the statement without it and see what happens.

Let's say you want random numbers from 1 to 100. Type

```
10 PRINT INT (RND * 100) + 1
```

```
20 GO TO 10
```

RUN it for a series of numbers. You will find this function more useful later.

Practice with **INT**:

```
PRINT INT 2.1
```

```
PRINT INT 301.789
```

etc.

TRIGONOMETRIC FUNCTIONS

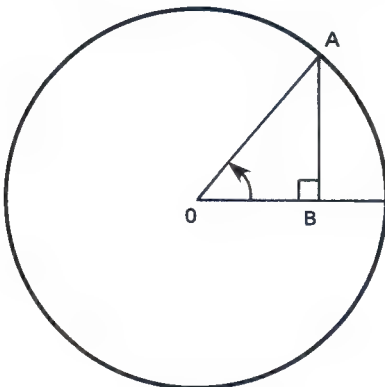
Don't panic! If you are beginning to think programming requires a knowledge of fancy mathematics—*don't!* Most programming will not use these tools; however, they are provided in your Timex and you should be aware of them.

If you don't understand what sine and cosine are all about, that's fine; but do run the short programs in this part of Level Four. Later when we draw a clock on the screen using the circle formula in Timex BASIC, you will at least realize the basis for the circle's creation.

Remember, you can skip this section with no problem; however, it is to your benefit to at least run these programs. Later on you might wish to return for a better understanding.

These functions are important if you want to draw circles, curves, graphs, etc. The three major trigonometric functions are sine, cosine, and tangent. For the nonmathematician it would be of value to understand what these functions are. These functions are determined from the right triangle and circle.

The *sine* of an angle is defined on a right triangle where one angle is always 90° (a perfect quarter-turn). If you draw a circle and put two radius arms equal to 1.0 on it—one horizontal, the other anywhere—we can study the sine function. Let's say the angle separating the two lines is 30°. Then draw a line from the cir-



cle at point A down to the horizontal line so it makes a 90° angle, thereby becoming a right triangle—AOB.

The *sine function* is the ratio of line AB to line AO. As point A moves around the circle, length AO is fixed—only AB changes. So you can see the sine function is a function of the angle AOB. When it is 0° , AB is 0. When it is 90° , AB is the same as AO and the ratio of AB to AO is 1. As the angle becomes 180° the value of AB decreases to 0. When you look at angles between 180° and 360° you find AB is negative. A plot of the value of AB as it goes from 0 to 1 to 0 to -1 to 0 again as it goes around the circle is called the *sine curve*.

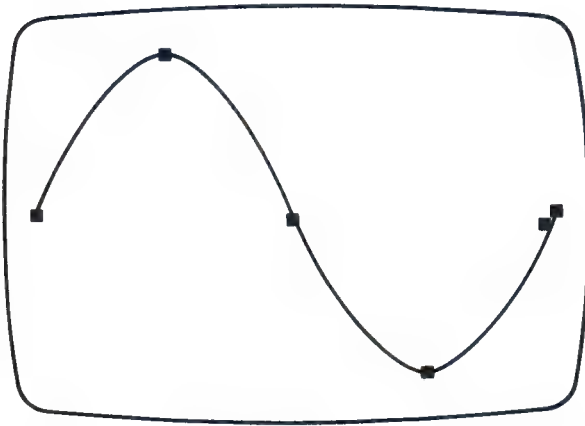
RUN this program and you will see what a sine curve looks like:

```
10 REM "SINE CURVE"
20 FOR N = 0 TO 63
30 PLOT N, 22 + 20 * SIN (N/32 * PI)
40 NEXT N
```

(**REM** is on the **E** key.)

Note **PLOT** is a keyword on the **Q** key. You can see why it is called a sine wave and is important in many calculations.

FIGURE 4.1



The *cosine function* is similar, but in our triangle it is defined as the ratio of OB to OA. As you can see on the circle, when the angle separating these two is 0° , the ratio is +1; as the angle grows to 90° OB becomes 0 and the ratio is 0/1 or 0. Up to 180° it becomes -1. From 180° to 270° it again becomes 0 but remains negative. At 270° it becomes positive and grows to 1. A curve of the cosine function is defined by this program. **RUN** it:

```
10 REM "COSINE CURVE"
20 FOR N = 0 TO 63
```



```
30 PLOT N, 22 + 20 * COS (N/32 * PI)
40 NEXT N
```

Note the similarity to the sine curve. You can see the comparison if you type in:

```
35 PLOT N, 22 + 20 * SIN (N/32 * PI)
```

Now **RUN** it!

What about the *tangent function*? This is the ratio of AB to OB. This is more interesting, for at 0° , AB is 0 and OB is 1 and the ratio is $0/1$ or 0. At 45° , $AB = OB$ and $AB/OB = 1$. At 90° , AB is 1, OB is 0, and the ratio is $1/0$ or *infinity*. At 135° , $AB = 1$ and OB is -1 , so the ratio is -1 . At 180° , $AB = 0$ and $OB = 1$; the ratio is 0 again. You can figure out what happens up to 360° .

The tangent curve is displayed by this program:

```
10 REM "TANGENT CURVE"
20 FOR N = 18 TO 45
30 PLOT N, 22 + 5 * TAN (N/32 * PI)
40 NEXT N
```

Note that it was only plotted for values of N between 18 and 45 because the tangent of 90° yields a number beyond any computer.

Hence, the COS, SIN, TAN functions are ratios of the sides of a standard right triangle and are always fixed for a given angle. Find $\sin 45^\circ$. You can take it off the sine curve you plotted or you can use

```
PRINT SIN (45/180 * PI)
```

Note that your trig functions require angles in radians (parts of a circle). A circle has 2π radians. Remember the formula for distance around a circle is 2π times the radius. Well, a circle of 360° is equivalent to 2π radians. So π radians is one-half of 360° or 180° .

Rule: To convert angles to radians multiply by π and divide by 180° .

Hence 45° equals $45/180\pi$ or $1/4\pi$ radians.

The other trig functions—secant and cosecant—are not separately included as they are merely reciprocals of the cosine and sine functions.

Functions which are sine, cosine, and tangent in reverse are *arcsin*, **ASN**, *arcos*, **ACS**, and *arctangent*, **ATN**. These are followed by a number for which you will get the angle in radians. To convert to degrees you multiply by 180 and divide by π . Hence given a ratio of 0.5000, the angle which would result in a sine of 0.5000 is 30° :

```
PRINT (ASN 0.5000) * 180/PI
```

If you are familiar with all the trig functions, go ahead and test the accuracy and capability of your Timex/Sinclair or ZX-81!

OTHER FUNCTIONS

In this section we shall briefly review square roots, exponentials, absolutes, and signs. Although these concepts are not necessary for most programming, you will encounter them and should have some familiarity with them.

SQR means square root. Try

```
PRINT SQR 25
```

You should get 5.

You might note that in the Timex BASIC dialect, functions do not require parentheses and that functions in an equation have priority over the basic arithmetic functions. But be careful: **TAN** N/2 is not the same as **TAN**(N/2).

EXP means exponential function, and **LN** means the natural log (to base e , not base 10). Try

```
PRINT LN 2.718281828
```

You should get 1, since that is the value of e . **EXP** is the inverse of **LN** and is based upon e .

Find the log to base 10 of 10:

```
PRINT LN 2/LN10
```

For any log to the base 10, simply divide the natural log of the number by the natural log of the base you are interested in. Try:

```
EXP 1
```

You should get the value of e .

The natural log of a number is the exponent of the number e , which would result in that number if you took e to that power. This means if you took $e ** 5$ the **EXP** of the result would be 5.0.

More useful functions include **ABS** and **SGN**. **ABS** on a number changes it to a positive value. It stands for absolute value. A number already positive will have no change.

```
PRINT ABS -4.1
```

You will get 4.1

Note again that most BASIC require parentheses; Timex dialect does not.

Finally the **SGN** or signum function will tell you only the sign of a number or variable. Try:

PRINT SGN —4.2

This is of value if you only want to know whether a value is positive, negative, or zero.

Practice with all these functions to become familiar with them. They will be helpful in programs later.

GRAPHICS SYMBOLS

The Timex has 22 graphic symbols built in. Each has a code number which you can determine directly with the **CODE** function. Basically there are three shades—white, black, and grey. Grey is depicted by grains of black and white. Properly arranged, these shades give you a wide range of possibilities of creating bar charts, patterns, or pictures. The smallest mark is one-fourth a full graphic symbol called a *pixel*. Pixels are used in graphing.

PLOT and **UNPLOT** of pixels is based upon two coordinates—X and Y (horizontal and vertical). Whereas lines are numbered from the top down 0 to 21, Y pixels are numbered 0 to 43 *from bottom to top*. X pixels are numbered 0 to 63 from left to right (similar to columns, only twice as many).

Refer to the display chart in Appendix B to help you see what you are doing.

With the pixel coordinate system, you can identify any spot on the screen with two numbers X and Y. To place a dot on the screen, you say **PLOT X, Y**, and to remove any dot, you say **UNPLOT X, Y**.

X and Y can be integers or any number including variables, functions, etc. For example, let's look at the sine curve program:

```
10 FOR N = 0 TO 63
20 PLOT N, 22 + 20 * SIN(N/32 * PI)
30 NEXT N
```

Here X is N, and Y is represented as a function of N. For $N = 0$ we plot a point on the $X = 0$ line at the left of the screen. The Y point is a function of X. For $X = 0$, $\sin 0$ is 0 and Y becomes 22. This places the first dot halfway from the top. This was done because it is known that the sine function can get negative. Therefore, the $X = 0$ point was located above the $Y = 0$ point by putting the 22 in the **PLOT** statement. The 20 in the plot statement was chosen to insure that the highest point of the curve is still on the screen—i.e., when sine of the angle = 1. The angle will range from 0 to 360° or 2π in order to see the whole curve. So if the end of the screen is to be 2π it would have to be divided

by 32. The maximum X value of 64 divided by 32 would yield 2 **PI**. Hence the preceding formula!

Now that we have an idea of how to develop a curve to fit the screen, it is time to learn how to put in axes and labels.

Prior to plotting a curve, input your X axis thus:

```
10 FOR C = 0 TO 31
20 PRINT AT 11, C; "-"      (dash—use minus sign)
30 NEXT C
```

and your Y axis thus:

```
40 FOR L = 0 TO 21
50 PRINT AT L, 1; ")"      (parenthesis on "0" for vertical line)
60 NEXT L
```

and label Y axis for the sine curve with:

```
70 PRINT AT 11, 0; "0"; AT 0,
0; "+1"; AT 21, 0; "-1"
```

and label X axis for the sine curve with:

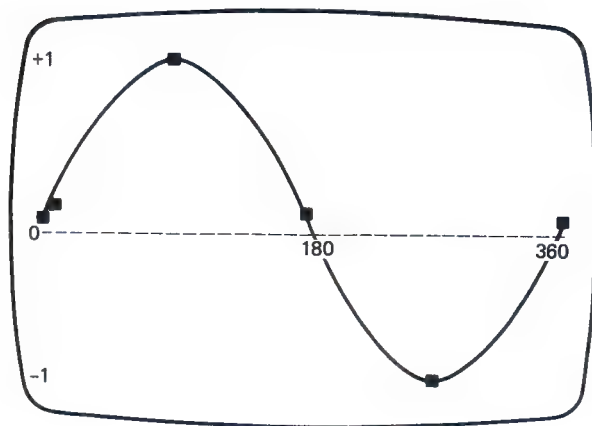
```
80 PRINT AT 12, 15; 180; TAB 29; 360
```

Be careful—**AT** statements use lines and columns, not pixel coordinates! So for a labeled sine curve filling your screen **RUN** this program:

```
01 REM "SINE CURVE"
05 REM PROVIDE AXES
10 FOR C = 0 TO 31
20 PRINT AT 11, C; "-"
30 NEXT C
40 FOR L = 0 TO 21
50 PRINT AT L, 1; ")"
60 NEXT L
65 REM PROVIDE LABELS
70 PRINT AT 11, 0; "0"; AT 0, 0; "+1"; AT 21, 0; "-1"
80 PRINT AT 12, TAB 15; 180; TAB 29; 360
90 FOR N = 0 TO 63
100 PLOT N, 22 + 20 * SIN(N/32 * PI)
110 NEXT N
```

Note: With 1K capacity you might overflow with an error 4. If so, command **CONT** to display balance of curve.

FIGURE 4.2



You can graph any function—try **SQR** in the above:

```
100 PLOT N, 20 * SQR (N/16)
```

Note: Your axes labels will be wrong and should be changed.
You can even try some random dots:

```
100 PLOT INT(RND * 64), INT(RND * 44)
```

If you get an error code, ask it how many dots it printed by putting in a command:

```
PRINT N
```

Try a simple quadratic equation $Y = X^2$:

```
90 FOR X = 0 TO 7
100 PLOT 9 * X, X ** 2
110 NEXT X
```

You will get a B error code because $7 ** 2$ is 49, which is off the screen.
Erase lines 1, 5, 55, 70, and 80.

We can now combine the sine and cosine functions and create a *circle*:

```
90 FOR T = 0 TO 60
100 LET A = T/30 * PI
110 LET SX = 31 + 18 * SIN A
120 LET SY = 21 + 18 * COS A
```

```
130 PLOT SX, SY
200 NEXT T
```

You can vary the size and shape of a circle (ellipse) by changing one or both of the "18" representing the radius, and the location by changing the 31 and 21 representing the center as the **PLOT** (31, 21) point. Now *erase* lines 10, 20, 30, 40, 50, and 60.

We can make this more interesting by printing numbers inside the above circle and calling it a clock!

Add the following statements and **RUN** the whole thing:

```
10 FOR N = 0 TO 12 STEP 3
20 PRINT AT 11 - 8 * COS(N/6 * PI), 15 + 8 * SIN(N/6 * PI); N
30 NEXT N
```

Doing the hands of the clock is possible but complicated for it requires your extended memory and will be reserved for an advanced text.

But you can still go one step farther and simulate a real time clock.

First we shall add to our vocabulary the command **PAUSE** (on the M key).

TIME CONTROL

The **PAUSE** statement is tied to the 8K ROM chip and the television picture. The screen prints a picture at 60 frames per second. **PAUSE** tells the computer to stop computing for a length of time based upon the number of frames elapsed.

Hence **PAUSE 60** says stop for 60 frames or 1 second before continuing to the next statement in your program. **PAUSE 30** would be about half a second. **PAUSE 600** would be 10 seconds. The largest workable pause is **PAUSE 32767** or about 11 minutes. Any number after that is an indefinite **PAUSE**. It is not quite a **STOP** because depressing any key will break the pause—putting it totally under your control.

Practically speaking, reacting to the command itself takes time; therefore, a true second might only be **PAUSE 52**. This you will have to experiment with. If you have sufficient memory, the timing will be fairly consistent. (Shortened memory squeezes the program!)

Now let's see a practical application. Add these lines to your circle program:

```
150 PAUSE 52
155 POKE 16437, 255
160 UNPLOT SX, SY
```

Note that line 155 is necessary when operating in **FAST** mode in order to prevent crashing of the program. A program is crashed when the contents in memory are shifted about and the listing is no longer intact. (**PEEK** and **POKE** will be more fully explained in Level Six.) The ZX80 8K ROM did not fully convert to

the ZX81 in that it could not achieve **SLOW** speed. This manual has thus far not given any consideration to what speed you are in, as it hasn't really made any difference.

RUN the clock program. You will note that it will tick away at a rate of about once a second. The value of T in the program was limited to 60—or 60 seconds. T can be set to 1000 seconds or even infinite time to provide a continuously working clock. If, however, it stops due to an error message, you can use the **CONTInue** command or **PRINT T** to see how many seconds it ran.

You will note that in **FAST** mode your screen will flicker—this is normal, due to the nature of the Sinclair central processing unit (CPU) being shared by computation and display features. You can overcome this in your Timex by giving first the command **SLOW** before you **RUN** the program. Generally, the Timex is normally on **SLOW**. To perform computations more quickly, it is necessary to give the **FAST** command after you turn on your unit and request **SLOW** when needed.

You may incorporate the **SLOW/FAST** command within a program by using it as a keyword. Thus, you can benefit from both features—**FAST** computation and **SLOW** display.

Although your Timex is normally in **SLOW** mode, you can type in the **SLOW** or **FAST** commands anytime like so:

Depress **FAST** followed by **ENTER**

You remain in this mode until directed otherwise by another command or from within a program as a line statement:

60 **FAST**

Look at the Interest Stimulator Number 2 of Level Two where we ran through a program at both speeds. Try some of the programs you have studied so far in both speeds to see the effect. **SLOW** was developed to minimize the flickering effect when depressing the **ENTER** key.

While your clock is ticking away you can stop it any time by pressing the **BREAK** command key (the **SPACE** key). It will stop and you will get a D error code.

Before going on to the next section, please depress **NEW** to clear the computer.

OTHER USEFUL TOOLS

SCROLL (on the B key) moves the display on the screen up one line. Try this program to see how it works:

10 **SCROLL**

20 **INPUT A\$**


```
30 PRINT A$
40 GO TO 10
```

(This is one of those loops it's impossible to get out of without turning off the unit.)

Try this program, inputting your first name:

```
10 INPUT A$
20 FOR I = 1 TO 100
30 PRINT TAB I/5; A$
40 IF I > 21 THEN SCROLL
45 PAUSE 20
50 NEXT I
```

INKEY\$ (on the B key) is a function that has no argument. It stands alone—it simply reads the keyboard. It becomes whatever key you are pressing in the **L** mode, including the **SHIFT** feature. It assumes the **L** mode. With a simple program you can turn your computer into a mini-word processor. It will print directly without requiring you to depress **ENTER** each time! You were given such a program in Level Three as an interest stimulator.

```
10 IF INKEY$ <> " " THEN GO TO 10
20 IF INKEY$ = " " THEN GO TO 20
30 PRINT INKEY$
40 GO TO 10
```

(Note: <> is on the T key—you cannot combine symbols on keys N and M.) Double quotes (" ") are typed as two individual quote marks. This works well on **SLOW**. If you have the ZX80, add:

```
32 PAUSE 42
33 POKE 16437, 255
```

If running in the **FAST** mode (such as on the ZX-80) this one will also work:

```
10 PAUSE 400000
20 POKE 16437, 255
30 PRINT INKEY$;
40 GO TO 10
```

In any case the **SPACE** key will act as the **BREAK** key and get you out of this loop. So to provide a spacer, you will have to program another key such as the **ENTER** key. Add these lines:

```
25 IF INKEY$ = CHR$ 118 THEN GO TO 50
50 PRINT "#";
55 GO TO 10
```


Note: 118 is the code for the **ENTER** key.
You now have a mini-word processor!

A GUESSING GAME

Now we can combine these new terms into a little game. We will have the computer display a number. Initially it is set to give you 1 second to respond by typing it back. If you succeed, the time allowed will decrease; if not it will increase. Your score is based upon how fast you can respond to the computer. When you feel you are fast enough, push the **Q** key for your score.

Let's write the program:

1. Let's pick a random digit:

```
30 LET A$ = CHR$(INT(RND*10) + 28)
```

2. Allow it to display for a variable time T:

```
35 PRINT A$
```

```
40 PAUSE T
```

```
50 POKE 16437, 255
```

3. Initialize the value of T:

```
10 LET T = 100
```

4. Have the T/S analyze the key you depress:

```
60 LET B$ = INKEY$
```

5. Compare the key you depress with the random digit and modify response time accordingly:

```
80 IF A$ = B$ THEN LET T = T * .9
```

```
90 IF A$ <> B$ THEN LET T = T * 1.1
```

6. Send it back to the beginning, scrolling the digit off the screen and displaying another:

```
100 GO TO 20
```

```
20 SCROLL
```

7. Include an allowance to request a score:

```
70 IF B$ = "Q" THEN GO TO 200
```

```
200 SCROLL
```

```
210 PRINT "YOUR SCORE IS #"; INT (500/T)
```

8. Now **RUN** this program and play the game. Note the function of the **SCROLL**, **PAUSE**, and **INKEY\$** keys! The faster you are, the higher the score.

DRAWING A BAR CHART

You are given the following data: Beginning in 1970 you earned for each year \$10,000 with 10 percent increase each year. Your real income, however, only increased 6 percent. (Real income means money inflation-adjusted so that it is equivalent to 1970 dollars.)

Show this on a bar chart for 5 years, comparing real and actual incomes.

SOLUTION

```

10 REM ESTABLISH INITIAL VARIABLES
20 LET Y = 1970
25 LET I = 10000
30 LET R = I
35 REM PRINT LEGEND AND LABELS
40 PRINT AT 5, 22; CHR$ 10; " = INCOME"
45 PRINT AT 10, 22; CHR$ 131; " = REAL INC"
50 PRINT AT 21, 12; "$10K"; TAB 22; "$20K"
55 PLOT 24, 2
56 PLOT 44, 2
60 REM BEGIN OUTER LOOP FOR EACH YEAR
65 FOR N = 1 TO 5
70 LET Y = Y + 1
75 LET X$ = STR$ (Y)
80 PRINT AT 4*N, 0; X$ (3 TO)
90 LET I = I * 1.1
100 LET R = R * 1.06
110 REM BEGIN INNER LOOPS TO PLOT FOR CHARTS
120 FOR J = 2 TO R/1000
130 PRINT AT 4*N - 1, J; CHR$ 138
140 NEXT J
150 FOR L = R/1000 TO I/1000
160 PRINT AT 4*N - 1, L; CHR$ 10
200 NEXT L

```

210 NEXT N RUN

Note: To **RUN** this on the ZX81 with only 1K delete lines 10, 35, 60, 110.

FIGURE 4.3



Please observe the following in the program and your results: Lines 20 to 30 established the starting point for the data. Lines 65, 70, 90, and 100 involve a loop to establish a set of data for each year for 5 years. Lines 40, 45, 50, 55, 56, 75, and 80 are merely label makers. The legend was placed over in an area where the graph would not extend to. Lines 75 and 80 were put into the loop since they print out the year as it is developed. Note the special **STR\$** function and the **X\$ (3 TO)** trick. These have not yet been covered and are not really necessary here. The function simply makes it print only the last two digits of the year after it changed the year from a numeric value to a string value. More on this in Level Five.

Of special value here are lines 130 and 160, for these actually print the bar chart. Character numbers were used for simplicity in typing. You could just as easily, with fewer bytes in fact, have inserted the graphic symbol in quotes. The **I** loops are necessary to repeat the graphics from start to the peak values calculated.

Note that **CHR\$ 138** is also a **SHIFT F** in Graphics **G** mode. It is half-grey and half-black. **CHR\$ 10** is **SHIFT S** and is half-grey and half-white.

Notice that we plotted the two-“color” graphic for the lower value (real income) since one shade represents real income and the other shade inflated income. When the maximum of the lower value was reached, it was no longer necessary to chart the black shade. Hence, the balance of the chart is done in grey only since the white side is invisible! Note also the logic in choosing the shifted **S** over the shifted **D**!

You can let your imagination go on this one!

The best method to learn programming is by *doing*. Experiment with this program by changing variables, graphs, labeling, etc. Once you understand how a program works you can begin writing on your own.

For this reason it is important you do all the exercises at the end of each level. They are there for both practice and instruction on additional little tricks which often save you considerable headache.

ARRAYS

One of the most frustrating experiences you've probably encountered so far is that of losing data.

In Exercise 2.25 involving commissions from Level Two, we had to manipulate the **PRINT** statement to insure all the results were properly displayed. Did you try to use the **GO TO** statement and find only one line of data results was printed? The computer only remembered the *last* value it had for the variables. The only way to repeat the total results was to **RUN** and reinput the data again.

So now it is time to learn how to work with data so that the computer remembers it forever (or at least until you turn it off) and how to retrieve it at will.

The technique involves the **DIMENSION** command and the process of setting aside "cubicles" of data for storage. Incidentally the computer does this anyhow for everything, but it is often moved or changed according to the instructions given. Dimensioning allows you to set aside *fixed* rooms in BASIC language, so you can readily fill and empty these rooms and manipulate the data. The one drawback is that the dimensionalized data automatically consumes 5 bytes each. So even with 48K of memory you only have at most 9000 slots for numeric data. This isn't much when you consider some businesses with large parts inventories, where each item might take several slots each: for part number, cost, sale price, quantity, description, etc. You can see why large computers are necessary in some areas such as student college records. Often this problem can be circumvented by utilizing ready-access external disk storage. Such a storage medium utilizing so-called floppy disks is available for the Timex and can retain 100K bytes on each disk.

The syntax rules for numeric arrays differs from those for string arrays. Hence, we shall reserve our discussion of string arrays for the next level, which shall concentrate on strings.

Numeric arrays are named with a single letter combined with a numeric "subscript," like so: **A(2)**. A numeric variable called **A** is not the same as a subscripted numeric variable **A(N)**.

Let's assume you have a series of room numbers 101 through 120. You can name them $R(1) = 101$, $R(2) = 102$, $R(3) = 104$, etc. $R(N)$, where N will vary from 1 to 20, is a *single-dimension numeric subscripted variable*.

If you tell the computer to set aside 20 spaces for this variable $R(N)$, it will be able to store the values in "cubicles." The direction in the program which tells it to allocate space is called the *dimension statement*.

10 DIM R(20)

It is always at the beginning of your program along with your initialized constants. The allowable space is only dependent upon your memory capability. Now depress **CLEAR** and then try this command on your unit:

DIM R(160)

If you have the ZX-81 with 1K you will get an error code 4. Try **DIM R(159)** and you will get no error.

If you have the Timex 10000 with 2K, you will not get an error code until you try to enter about **DIM R(375)**. Remember to depress **NEW** before using this command to insure maximum memory space is available.

If you have the extended 16K pack, try various ranges for **DIM** and you will be surprised at how much more space you have.

Remember too, at the smaller memory capacities a larger proportion is used for the program itself, and your actual data space is considerably less.

Let's fill up the array R(10) with these "room numbers" using a loop:

```
05 DIM R(20)
10 FOR I = 1 TO 20
20 LET R(I) = 100 + I
30 NEXT I
RUN
```

This data is now in your computer. To prove it just try:

PRINT R(4) ENTER

You will get the number stored in that location or 104. You can have it **PRINT** all of them. Delete your program—*not* with **NEW**, for it will reset your variables to 0, but by entering the line numbers only. The variables are now in storage.

Now type this program and **GO TO 10**:

```
10 FOR N = 1 TO 20
20 PRINT R(N),
30 NEXT N
```

Note that an array still requires a **PRINT** control of some kind when you make it display. Once an array is in storage do not **RUN** your program. Use **GO TO** command instead, since **RUN** clears all your variables to zero!

The advantage of an array is that you can create a table of data by using a two-dimensional array. Think of two dimensions as a grid or as columns and rows.

The dimension statement A(5,5) sets aside 25 spaces for data (5×5). True, you could have used A(25), but the idea of an array is to put things in *order*.

Here's what A(5,5) means: 5 rows of 5 columns of data named as follows:

A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)
A(4,1)	A(4,2)	A(4,3)	A(4,4)	A(4,5)
A(5,1)	A(5,2)	A(5,3)	A(5,4)	A(5,5)

Given the same method as on single arrays using loops, let's fill the spaces with some numbers:

```

10 DIM A(5,5)
20 FOR I = 1 TO 5
30 FOR J = 1 TO 5
40 LET A(I,J) = I * J
50 NEXT J
60 NEXT I
RUN

```

FIGURE 4.4

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

0/110

Now erase your program. Then run this one (use **GO TO 70**):

```

70 FOR I = 1 TO 5
80 FOR J = 1 TO 5
90 PRINT TAB 7 * J - 7; A(I, J);
100 NEXT J
110 NEXT I

```

Rule: An array subscript cannot be \emptyset . If you name a two-dimensional array **A**, you cannot have a single-dimension array also named **A** even though you may have a numeric variable named **A**.

Can you have a three-dimensional array? Sure! Try **DIM B(2,2,2)**. This means you could have perhaps 2 pages, 2 columns, 2 rows—or whatever.

They would be called B(1,1,1), B(1,1,2), B(1,2,1), B(1,2,2), B(2,1,1), B(2,1,2), B(2,2,1), B(2,2,2). This consists of 8 locations ($2 \times 2 \times 2$).

You can fill these with 3 nested loops. Try this program and see how it works:

```

5 DIM B(3,3,3)
10 FOR I = 1 TO 3
20 FOR J = 1 TO 3
30 FOR K = 1 TO 3
40 LET B(I,J,K) = I * J * K
50 PRINT TAB K * 9; B(I,J,K);
60 NEXT K
70 NEXT J
80 NEXT I

```

There is no limit to the number of dimensions in a numeric array other than memory space.

Let's look at some practical applications of arrays to better understand their use.

So far for demonstration purposes we automatically computed some values to store. But in reality we often store given data which must first be input individually in the array and removed as required.

Review Exercise 2.25 of Level Two and its solution in Appendix A.

STORING AND RETRIEVING DATA

We used a loop to input data, compute results, and print it all out—for one-time use only. Now we wish to make it permanent. First we allocate all the variables into a grid by adding the following statements (you can type in the text solution and edit lines as we go along):

```

11 DIM N$ (3,10)
12 DIM I$ (3,4)
13 DIM S(3)
14 DIM R(3)
15 DIM Q(3)
16 DIM P(3)
17 DIM T(3)
18 DIM C(3)

```

Note that we could not use S1(3), S2(3), etc. due to syntax rules—so we renamed them.

For ease, here is the program from Level Two:

```

5 LET J = 0
20 PRINT AT 5, 0; "SALESMAN"; TAB 11; "I.D."; TAB 16; "SALES";
TAB 25; "COMM"
25 FOR E = 1 TO 3
30 INPUT N$
40 INPUT I$
50 INPUT S1
60 INPUT S2
70 INPUT S3
80 INPUT S4
90 LET T = S1 + S2 + S3 + S4
100 IF T > 1000 THEN LET C = T * 0.15
110 IF T < 1000 THEN LET C = T * 0.10
120 LET J = J + T
130 PRINT AT 6 + E, 0; N$; TAB 12; I$ TAB 17; T; TAB 25; C
135 NEXT E
145 PRINT AT 11, 1; "TOTAL COMMISSIONS PAID: #$$"; J

```

Input data from Level Two was:

Name	I.D.	SALES			
		1	2	3	4
T. Crivaro	120	45	2007	314	105
P. Silvers	683	15	31	9	18
J. Rothberg	314	29	91	54	50

Now change lines 30 through 80 accordingly:

```

30 INPUT N$(E)
40 INPUT I$(E)
50 INPUT S(E)
60 INPUT R(E)
70 INPUT Q(E)
80 INPUT P(E)

```

Now modify computation lines as follows:

```

90 LET T(E) = S(E) + R(E) + Q(E) + P(E)
100 IF T(E) > 1000 THEN LET C(E) = T(E) * .15
110 IF T(E) < 1000 THEN LET C(E) = T(E) * .10
120 LET J = J + C(E)

```


Now the **PRINT** statement can be simpler:

```
130 PRINT N$(E); TAB 12; I$(E); TAB 16; T(E); TAB 25; C(E)
```

With 1K delete lines 20, 130, and 145 (embellishments).

RUN the whole program as corrected and provide inputs. Erase the program *except* lines 25, 130, and 135 (1K units—*now* type in line 130), then say **GO TO 25**. Your results will still be printed as they are stored.

Now **CLS** and ask for the commission earned by Mr. Rothberg and the total commissions paid:

```
PRINT N$(3), C(3), J
```

Note that you used some *string arrays* in this problem. They were basically similar in this case to the numeric arrays. Note the use of a second dimension—this will be explained further in Level Five.

Some BASIC automatically assumes a 10×10 array when using subscripted variables. Your Timex dialect does *not*. You must always **DIM** these variables before you can use them. It is not necessary to fill all the “cubicles,” but if you do not have enough you will be unable to store all your data.

You have tried an input/output array using loops to request the input and loops to display the required output. An array, however, has an even more practical function—that is, to contain an assortment of data which you can retrieve as needed.

So let's try a complicated pattern, store it as data, and then retrieve it as necessary to show how this works. Prior to this a few words on the **LOAD-SAVE** feature of your system. Review your handbook for information on storing programs on your cassette tape recorder. To save a program you must name it with a **REM** statement in quotes, then enter in the command **SAVE** followed by your program name in quotes. Start the tape recorder and then press the **ENTER** key. Once the screen returns with 0/0 code, your program has been saved. Then press **NEW** and **ENTER** to clear the computer—if you dare! Now call your program by entering **LOAD** followed with your program name in quotes. Start the tape on **PLAY** and depress **ENTER**. Your Timex will find the program and, when the screen is clean, stop your recorder. Press **LIST** and you will see your program (hopefully).

Before you try this exercise, make sure this feature works well. If you are having trouble, consult your manual. (Note: **LOAD** is a keyword on the J key and **SAVE** is on the S key.)

We are going to establish a table of data. Most institutions have salary schedules based upon position and seniority. The federal government service pay scale, for example, ranges from GS level 1 to GS level 15—each level with 10 graded steps. Each step consists of an hourly rate. For our example, we shall assume the following levels, each with 5 steps for your small business:

SALARY BY RATE/LEVEL					
Level/step	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>
L1	4.70	4.95	5.10	5.60	6.10
L2	5.00	5.50	6.00	6.66	7.01
L3	5.90	6.40	6.95	7.45	8.03

Your immediate objective is to store the above grid or matrix into memory by using a dimension statement for future use. Include a **PRINT** statement to verify it is properly inputted. To lay out your **PRINT** statement, it helps to use graph paper—see Appendix B for a sample form.

The following program will file the above data in an array. When you **RUN**, it will advise you which information it is seeking by level and step number. **RUN** this program and provide information as required from the table:

```

5 REM "SALARY SCALE"
10 DIM S(3,5)
15 PRINT AT 10, 0; "LEVEL"; TAB 13; "STEPS"
20 FOR L = 1 TO 3
30 FOR R = 1 TO 5
40 PRINT AT 1, 1; "INPUT LEVEL #"; L; "STEP #"; R
50 INPUT S(L,R)
60 PRINT AT L + 11, 1; L; TAB(R*6-2); S(L,R)
70 IF L = 3 AND R = 5 THEN PRINT AT 1, 1; "#####
#####" (AND is on the 2 key)
80 NEXT R
90 NEXT L

```

Note: Lines 40 and 70 are optional. Line 70 is a logic statement which will clear **PRINT** of line 40 after it is no longer needed. Logic statements will be discussed in detail in the next level.

To keep this data, never **RUN** this program again. Now depress **CLS** to remove the chart from the screen. Then verify it is still there by deleting lines 40, 50, and 70 and entering the statement **GO TO 15**. If it appears, continue to erase lines 10, 15, 20, 30, 40, 60, 80, and 90.

Now **SAVE** this program: "SALARY SCALE".

So you now have on tape the variable S(L,R) under the title "SALARY SCALE" available when you need it. Now be brave and clear your computer with **NEW**.

Now assume you need to calculate your personnel's pay for the week. You are given their hours worked and their pay scale as follows:

Johnson	Grade 2	Step 3	40 hours
Roberts	Grade 3	Step 4	36 hours
Davis	Grade 1	Step 1	42 hours

We write a program to obtain the data file and print out their names and gross pay. First **LOAD** into your computer "SALARY SCALE" and then put in this program. Do not **RUN** it. Initiate it by a **GO TO 10** statement to save those dimensioned variables.

As the program calls for you to input information, provide it from the above chart. It will automatically retrieve the wage rates from your stored array S!

```

06 REM COMPUTING GROSS PAY
10 DIM N$(3)
14 DIM P(3)
15 FOR I = 1 TO 3
20 PRINT AT 1, 1; "INPUT EMPLOYEES NAME #####"
   ##"
25 INPUT N$(I)
30 PRINT AT 1, 1; "INPUT GRADE LEVEL FOR #"; N$(I)
35 INPUT L
40 PRINT AT 1, 1; "### INPUT STEP RATE FOR #"; N$(I)
45 INPUT R
50 PRINT AT 1, 1; "INPUT HOURS WORKED BY #": N$(I)
55 INPUT H
60 LET P(I) = S(L,R) * H
70 PRINT AT 10 + I,0; N$(I), P(I)
80 NEXT I

```

Your results should look like this:

```

JOHNSON  240
ROBERTS  268.2
DAVIS    197.4

```

Hint: When trying to **EDIT** a line and it won't work, move cursor to the line, **CLS**, then depress **EDIT** and the line will appear in **EDIT** mode.

You are now ready for practice assignments. But first an interest stimulator.

INTEREST STIMULATOR

```

10 LET L = USR 2093
20 FOR A = 1 TO 100
30 NEXT A
40 GO TO 10
RUN

```

Wait 5 seconds. Depress **BREAK**, observe. Depress **ENTER**.

SUMMARY REVIEW

You have added the following to your BASIC vocabulary: **RND, RAND, INT, SIN, COS, TAN, ASN, ACS, ATN, SQR, LN, EXP, ABS, SGN, PLOT, UNPLOT, PAUSE, SCROLL, INKEY\$, DIM, LOAD, SAVE.**

You have learned how to manipulate math functions, draw curves and circles, create electronic die, prepare bar charts, and generally have fun.

You've seen how to create motion using the time control function. You can use your unit as a typewriter. You are familiar with saving programs on tape.

You are at home with subscripted variables and can use them to maintain variables in a file. You have expanded your ability to write programs and to interpret complicated ones.

EXERCISES

- 4.1 What is wrong with the following? **LET Y = INT(RND 6)**
- 4.2 *Problem:* On the first square of a checkboard (with 64 squares), you put a penny. On the second you put 2 pennies, then 4 on the third, 8, 16 and so on—doubling until you cover all squares. How much did this cost you?
- 4.3 **PLOT** a straight line: $Y = X$.
- 4.4 **PLOT** the function **LN**.
- 4.5 Given the program for a circle in this chapter, modify the program to give a circle with half the diameter.
- 4.6 Show half a circle on the screen.
- 4.7 Write a program which will give you six different lottery numbers between 1 and 36.
- 4.8 Modify the sine curve program so you see four sine curves.
- 4.9 Using the “**SALARY SCALE**” compute your annual salary (2080 hours per year) if you are on the highest step and level. Assume you cannot “see” the hour rate.
- 4.10 Is the following true or false? **C**, **C(I)**, and **C(I,I)** can coexist in the same program, and each is different.
- 4.11 **DIM** (10, 10, 10, 10) is a legitimate statement in Timex BASIC—true or false?
- 4.12 What would the result of this program be?

```
10 FOR N = 0 TO 63
20 PLOT N, 22 + 20 * SIN(N/32 * PI)
30 PLOT N, 22 + 20 * SIN((N + 32)/32 * PI)
40 NEXT N
```

- 4.13 How long will **PAUSE 72000** work? How do you stop the pause early?
- 4.14 Why does the function **INKEY\$** end in a dollar sign?
- 4.15 Can you use variables in a dimension statement?
- 4.16 What would be the result of each of the following?
- a. **PRINT COSA/SINA * TANA + 1** if $A =$ a real number
 - b. **ABS (X - Y)** when $X = 1, Y = 5$
 - c. **SQR 49 * 4**
 - d. **SQR (49) * 4**
 - e. **SQR (49 * 4)**
 - f. **SGN (X - Y)** when $X = 1, Y = 5$
- 4.17 What would happen here?
- 10 **PLOT X, Y**
 - 20 **UNPLOT X, Y**
- 4.18 What is wrong with **LOAD " "**?
- 4.19 Will **SAVE SALARY** work?
- 4.20 *Special problem:* Write a program to print a checkerboard such that it will fill most of the screen.

LEVEL FIVE

STRINGS AND LOGIC

In the course of your study of BASIC, you have briefly encountered STRINGS in **PRINT** statements and as variables. Your computer, though, has many additional and unique string handling capabilities which you will find as valuable tools. These involve several string functions, special dimension rules, substrings, and slicing techniques as well as being involved in logic statements.

Remember, a string means a string of characters, a character being any word, letter, or symbol. Strings are always depicted within quotes. When you assign a string to a variable, it is called a string variable. To alert the computer that the variable is a string, we identify a string variable with a letter and the dollar sign. Furthermore, a string variable can be dimensionalized like the numeric variable, via the dimension **DIM** statement. There are two important rules to remember with string variables.

Rule 1: In a program you cannot have a string variable and a subscripted string variable with identical names.

(This is contrary to the rule using numeric subscripted variables.)

Rule 2: The dimensionalizing of a string variable differs from the numeric variable in that it requires an additional dimension to describe it.

Remember, the **DIM** statement sets aside spaces for storing data. **DIM A(6)** means that the computer is to store 6 locations of numeric variables—5 bytes are automatically set aside for storing each number. Strings, however, are handled differently due to the nature of a string. Since strings can be of any length, the computer must be told how much space is needed. Hence, the **DIM** statement for a string variable might be represented as **DIM A\$ (5,10)**. This sets aside 5 locations for 5 variables each of a maximum length of 10 characters. When using one-dimensional arrays simply add a second dimension for strings. The second dimension will allow any string up to that length; if the string is longer it is truncated—cut short. When calling for the string variable the single dimension is all that is necessary—it will call out whatever is in storage.

String variables can be of several dimensions like the numeric subscripted variables. The two-dimensional string matrix would require a third dimension to allocate the length of the string. Hence, the statement **DIM(3,4,10)** is a 3×4 matrix, each element having a maximum of 10 characters.

Normal usage involves combinations of numeric and string subscripted variables. In any size matrix the string variables are dimensioned with the extra-character-length dimension. You can now better understand the matrix problem presented in Level Four.

STRING FUNCTIONS

Your computer has three functions which interplay between numeric and string variables. These functions—**LEN**, **VAL**, and **STR\$**—can be valuable tools in manipulating your computer to perform in games and special applications.

The **STR\$** function (Y key) essentially takes any number and changes it to a string. In other words, a real number becomes a character under quotes. This may occur when you want to treat a number with string tools by first changing it to a string so the computer will let you treat it as a string.

Hence, **STR\$ 2.345** changes it to “2.345”: It is no longer a numeric value but a string of digits and a period character!

The **VAL** function (J key) has the reverse effect: It gives the numeric equivalent value if the string were numeric. Hence **VAL “3.45”** would be 3.45 and **VAL “3 + 2”** would be 5 instead of 3 + 2.

Test the **STR\$** and **VAL** functions on your computer using the **PRINT** statement.

What happens if you try **VAL “HELLO”** or **STR\$ “2.45”**? You get an error code 2.

The third function, **LEN** (K key), is an interesting one: It gives you a numeric value which is equal to the length of a string variable. Hence **LEN "TODAY"** results in a number 5. This number is real. Try **PRINT LEN "TODAY" * 2.2**. You will obtain 11! Tremendous!

Practice with these functions until you understand what they are doing.

As with the other functions, you can use appropriate variables with these as well. For example, these statements are valid:

```
10 STR$ N or 10 STR$ N(I,J)
10 VAL N$ or 10 VAL N$(A)
10 LEN N$ or 10 LEN N$(A)
```

Of course, the variables themselves must be defined in their program somewhere before they are used.

A note about the **VAL** function: It has one important restriction—it must be listed first when part of a larger expression. That is, line 20 below is valid, while line 10 is *not*:

```
10 LET Z = 7 + VAL "X"
20 LET Z = VAL "X" + 7
```

This goes also for any statement using two coordinates such as **PRINT AT**, **PLOT**, or a subscript in a variable. Hence you cannot say

```
10 PRINT AT M, VAL "N"
```

You could, however, say 20 **PRINT AT VAL "M", N**

To make line 10 valid you would be creative to say

```
5 LET N = VAL "N"
10 PRINT AT M, N
```

Try this little program and see what it does:

```
05 LET A$ = "NO"
10 DIM N(5)
15 FOR L = 1 TO 5
20 LET N(L) = L
25 NEXT L
30 PRINT STR$ N (3)
35 PRINT LEN STR$ N(3)
40 PRINT N(LEN A$)
```

Your results should be:

3
1
2

Line 30 converts the third N (which was set to a 3) to a "3". (The quotes do not show up on the string.) Line 35 then says what is the length of that string: It has one digit, and therefore the second result is 1. Line 40 looked at the length of the string in line 5, which has 2 characters!

These string functions can also be combined in interesting ways with each other as string expressions:

```
PRINT VAL (STR$ LEN "7893")
```

This would be evaluated as **LEN** "7893" equal to 4; then the 4 is changed to a string "4" by **STR\$**, then back to the **VAL**ue of 4! Perhaps a little redundant?

Try combining this with arithmetic expressions:

```
PRINT VAL (STR$ LEN "TODAY" + "+4")
```

The result would be 9. In order to add the numeric 5 to a string 4, it was first necessary to change the numeric 5 to a string value to put them both within quotes—as "5 + 4". Normally two strings when "added" are only connected. This would make "5" + "4" come out as 54; since we have included the plus sign in the original quote it shows up as "5 + 4". The **VAL** function then allows the "5 + 4" phrase to be evaluated as if it were a numeric statement.

Addition or connection is the only valid arithmetic function you can perform on a string. For example, try **PRINT "BOOK" + "KEEPER"**. The other three functions of minus, multiply, or divide have no meaning with strings.

Furthermore, these string functions can be combined with numeric functions if properly done. For example, you could do:

```
PRINT ATN (VAL(STR$ LEN "1235" + "+5.378")) *180/PI
```

Result: An angle of 83.91°.

Note: When in doubt about which is performed first, use parentheses. A priority chart is provided in your computer manual for your information.

Of what value are these string functions? At first glance it is difficult to imagine how these are used. This is where computer programming verges on being an art as well as a science. It requires creative thinking and imagination to develop some of the unique programs you have encountered and will encounter. The more you program, and the greater your familiarity with your computer, the more creative ideas and techniques you can come up with.

Should you find programming a creative and fun adventure, it can become profitable for you to write and publish programs for sale for fellow Timex/Sinclair owners!

Getting back to these string functions. Suppose you want to establish a program to display math problems with solutions for a young student. You know an expression like $2 + 4$ would be written as 6 on the computer. You also know if you printed " $2 + 4 =$ " on the screen, the computer could not evaluate it. So isn't it fantastic that you can both print and evaluate that expression with the tools you have just learned?

Here's a very simple program displaying a problem with the result:

```
10 LET A$ = "1 + 3"
20 PRINT A$; "=?"
30 INPUT B$
40 PRINT A$; "=", VAL A$
```

In this program lines 10 and 20 establish the problem. Line 30 allows you to input your response. Line 40 prints the problem as well as the result:

1 + 3 = 4

Whereas the left side is a string expression, the right is the numeric equivalent. You can try other values for A\$. Try something more complicated. Change line 10 to

```
10 LET A$ = "(ATN 1 * 4) * 2/PI"
```

The result: 2.

You can probably see the value of using this in a math course.

In the above brief math program, line 30 was inserted primarily to entice you to wonder how you can get your computer to evaluate the student's response, comparing it with the real value and then grading the student. This will involve writing a program using both string expressions and logic expressions.

Before delving into the complicated subject of logic expressions, we shall continue with developing our understanding of string concepts, including substrings and slicing.

SUBSTRINGS

A *substring* is a series of consecutive characters of a string. In the string "SUPERMAN" the following are some substrings:

```
SUPER
MAN
ERM
E
M
UPERMAN
```

Notice that a substring is identical to the string but “sliced.” For example, if you sliced off the letters SUP in front of the E and the RMAN after it, you have left the substring “E”!

The notation to show this slicing is a set of parentheses with the word **TO** as follows:

`"ABCDEFGH" (2 TO 7) = "BCDEFGH"`

In the above expression, you have sliced off the first letter, A, leaving the balance as the substring. You can slice any portion of a string by properly designating the beginning and ending character within the parentheses. The numbers above say to produce the substring consisting of the second through seventh character of the string `"ABCDEFGH"`.

What happens if the digit is omitted as below?

`"ABCDE" (TO 5)`

Your computer automatically assumes the beginning or end if no numbers are present. Hence, the substring `"ABCD" (TO)` is the same as the string itself! In fact, if you want, you can write it as `"ABCD" ()` *without* the **TO** to represent the whole string as a substring.

You can cut your string down to one character by noting the position of the character. To slice **SUPERMAN** to a mere **E** you can write:

`PRINT "SUPERMAN" (4 TO 4)`

or more simply

`PRINT "SUPERMAN" (4)`

Can you see the value in this? Suppose in a game of Hangman you want to evaluate each character of a string. You may need an expression like:

`A$(N)`

where N could vary from 1 to the full length of the word.

Beware, this almost looks like a dimensioned subscripted string variable! This illustrates the important rule prohibiting the same name for a subscripted string variable and a regular string variable. Your Timex would get confused!

Other rules for slicing are as follows:

1. Your beginning and ending numbers must be within the length of the actual string *except* when both are *outside* the range, resulting in a null string (blank spaces). Line 20 is valid; line 10 is not:

```
10 PRINT "ABCDE" (3 TO 6)
20 PRINT "ABCDE" (7 TO 9)
```

2. You cannot use negative numbers within the parentheses:

```
PRINT "ABCD" (4 TO -1)
```

is invalid.

3. You can function a sliced string; for example:

```
PRINT LEN "ABCD" (2 TO 3)
```

is a valid statement.

4. Using the substring you can make changes within a string, that is, assign a new value to a portion of a string. Try this program:

```
10 LET A$ = "I LOVE YOU"
12 PRINT A$
15 LET A$ (3 TO 6) = "HATE"
20 PRINT A$
```

In line 15 we have replaced a portion of a string and created a new string!

Now what about the string within a dimension? Can we slice that also? If so, how? It obviously can't be the same technique, since A\$(3) would be the third string variable of an array and not the third character of the string.

Well, remember that in an array a second dimension is required to establish character locations.

Hence DIM A\$ (5,6) limits the string length to 6. But when you call for a variable such as A\$ (3) it will be up to length 6 as stored. If you called for A\$ (3,6), however, you would get the sixth character of the third string. You can also change that dimension to a slicer such as A\$ (3,2 TO 4), printing only the letters 2 through 4.

Consequently, with a two-dimensional matrix with string variables dimensioned as DIM B\$ (4,4,8), a call for variable B\$ (2,1) would result in a string of up to 8 characters in length and B\$ (2,1,1) would represent only its initial! Say the word is **MARCH** located at 2,1. Then B\$ (2,1) would print **MARCH**, whereas B\$ (2,1,4) would print C. What might B\$ (2,1,6) produce? Normally an error, but since its dimension was 8 it really was stored as **MARCH ###** (with three spaces) and therefore the sixth position is the space or a null string. The word **ARCH** would be represented by B\$ (2,1,2 TO 5).

Now that you have a grasp of words in BASIC, let's go on to some logic.

LOGIC

In general, logic expressions utilize the **IF . . . THEN** statement. Logic expressions are similar to numerical expressions but instead of using the arithmetic functions add (+), subtract (−), divide (/), and multiply (*), they use the three logical operators: **AND**, **OR**, and **NOT** along with the six relational operators:

Equal to (=)	on SHIFT L
Greater than (>)	on SHIFT M
Less than (<)	on SHIFT N
Greater than or equal to (>=)	on SHIFT Y
Less than or equal to (<=)	on SHIFT R
Not equal to (<>)	on SHIFT T

Note:

AND is **SHIFT 2**

OR is **SHIFT W**

NOT is **FUNCTION N**

The logic operands in an **IF** statement cause the computer to treat them as either *true* or *false*. If it finds a condition valid, it is deemed true and given the *value* or *bit* of “1”. If false, it is given the value of “0”.

To see this ask your computer to

```
PRINT 1 < 2
```

You will see a 1 appear on the screen—i.e., it is a true statement.

Now try

```
PRINT 0 > 2
```

The result was 0 or false!

When you make a statement like

```
IF X = 6 THEN GO TO 110
```

the computer evaluates the value of X and compares it with the numeric 6. If they are the same, it signals a true value. The **IF** statement is ignored when the condition comes out false or 0. When the condition is true or 1, the computer looks beyond **THEN** and acts upon the subsequent *keyword*.

Try this simple program testing your psychic power! Have someone input a number N and see how long it takes you to guess it:

```
10 INPUT N$
```

```

20 INPUT A$
30 IF A$ < > N$ THEN PRINT "WRONG"
40 IF A$ = N$ THEN GO TO 60
50 GO TO 20
60 PRINT "CORRECT"

```

Furthermore, you can use these logical operations and combine one or more conditions, including mixing of string variables. Try this program combining both:

```

10 INPUT N$
15 INPUT N
20 INPUT A$
40 INPUT A
50 IF A$ = N$ AND A = N THEN PRINT "GOOD"
60 PRINT "TSK TSK"
70 GO TO 20

```

So far we've used the "equals" sign " $=$ ". But what does it mean to say "greater than" or "less than" with *strings* since they are not exactly numbers?

It means it is before or after in terms of *alphabetical order*. This works because it compares the letters one by one. The letters are coded in the ROM chip in both numerical and alphabetical sequence. Therefore, the computer can compare their locations by comparing their respective codes. Hence it automatically alphabetizes.

Try this program:

```

10 INPUT A$
20 INPUT B$
30 IF B$ > A$ THEN PRINT A$,, B$
40 IF B$ ≥ A$ THEN PRINT B$; A$

```

Note the different effect using two commas versus the semicolon (in case you forgot). Lines 10 and 20 insert the two words you want in alphabetical order. Lines 30 and 40 will compare them and print accordingly. Try inputting various words to see this work. Note that you can only alphabetize two words so far. Further programming will be reviewed in a later chapter to enlarge this capacity into a large sorting ability.

Note that your **IF** statements are constructed like so:

IF condition **THEN** statement

This says a condition exists between the **IF** and **THEN** words followed by an action of some kind. The statement is necessary in Timex BASIC. You will find

other BASICs which might allow **IF** $X = 4$ **THEN** 110, meaning to go to line 110. Timex BASIC *requires* a keyword such as **GO TO**. But because of this minor limitation, it also allows you to follow it with keywords like **LET**, **STOP**, and **PRINT**, saving you time and space in the long run.

The “condition” between **IF** and **THEN** is simply either a truism or a falsity—not necessarily an equation or equality, as you have been accustomed to seeing. When you said

IF $X = 5$ **THEN GO TO** 30

you were really saying

IF ($X = 5$) **THEN GO TO** 30

or

GO TO 30 **IF** the value $X = 5$ is true.

Your condition could simply be:

IF X **THEN GO TO** 40

meaning: when X is not equal to 0; or :

when X is a number larger than 0, you can go to line 40. You can state the reverse by using the **NOT** function:

IF NOT $X = 5$ **THEN** . . .
IF NOT X **THEN**

are both perfectly valid. The condition now being evaluated includes the **NOT** function. In **NOT** $X = 5$ the condition is true when X does not equal 5.

In **NOT** X the condition is true when X is false! This can sure get confusing—so try not to use **NOT** except to make things clearer, since anything written with **NOT** can be written in another way without it.

BOOLEAN OPERATORS

AND and **OR** are known as *Boolean operators*. They behave with true and false concepts. Let's establish for these operators a “truth table,” i.e., a matrix listing the results of an operation given all possible true and false combinations of the variable.

Given two variables X and Y :

X	Y	X AND Y	X OR Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

This states that the statement **X AND Y** is true *only* if both X and Y are each true (1), and that the statement (**X OR Y**) is true whenever *either* X or Y is true (1).

These concepts can be added to the condition in an **IF** statement:

IF X AND Y THEN . . . or **IF X OR Y THEN . . .**

Note that no punctuation exists in that portion of the statement.

X and Y can be truth values; but they can also be logical expressions.
Hence:

IF X < 5 AND Y > = 8 THEN . . .

is a valid condition, as is

If X < 5 OR Y ≥ 8 THEN . . .

or this:

IF NOT (X < 5 AND Y ≥ 8) THEN . . .

or

IF NOT X < 5 AND Y ≥ 8 THEN . . .

or

IF NOT X < 5 OR Y > 8 THEN . . .

It all depends upon what you want to say!

Now let's practice with our Timex to demonstrate its capacity to deal with logic operators by writing short programs:

```

10 INPUT X
15 INPUT Y
20 IF X < 5 AND Y > 8 THEN GO TO 50
30 PRINT "FALSE"
40 GO TO 10

```



```
50 PRINT "TRUE"
60 GO TO 10
```

Run this program and input various values of X and Y including 0 and negative values. Set up a truth table—including X, Y, and your results. Use values of X and Y between 4 and 9.

Now replace line 20 with

```
20 IF X < 5 OR Y > 8 THEN GO TO 50
```

and **RUN** with various inputs of X and Y. (This is the best way to understand the logic your computer uses!) Note that line 20 uses **OR** instead of **AND**.

Repeat the program using

```
20 IF X ≤ 5 OR Y <> 8 THEN GO TO 50
```

RUN with various values of X and Y.

We can make the statement more complicated by adding variables:

```
5 INPUT J
20 IF X ≤ J OR Y <> J THEN GO TO 50
```

RUN this now varying J, X, and Y. (J is input only once for each **RUN**.) We can go further and create a complicated logic statement:

```
20 IF X < 5 AND (Y < 8 OR X <> 2) THEN GO TO 50
RUN
```

(Delete line 5 so you don't get confused.)

Again try different variables and note how you obtain "true" (when X is less than 5 and either X was not a 2 or Y was less than 8).

Try

```
17 INPUT Z
20 IF X ≤ 4 OR X ≥ 8 AND Y ≥ 4 OR Y = 2 AND Z <> 1 THEN
GO TO 50
RUN
```

See result a in the table below.

X	Y	Z	a	b	c	d
1	4	1	T	T	F	F
8	4	0	T	T	F	T
4	2	2	T	T	F	F
5	2	1	F	F	T	T
5	3	1	F	F	T	T
6	2	0	T	F	T	T

Given **AND** has priority over **OR**, if you **RUN** with the following, you should get identical results:

```
2Ø IF X ≤ 4 OR (X ≥ 8 AND Y ≥ 4) OR (Y = 2 AND Z < > 1)
RUN
```

Now what will this do with the same variables?

```
2Ø IF (X ≤ 4 OR X ≥ 8) AND (Y ≥ 4 OR (Y = 2 AND Z < > 1))
```

See result b in the table on p. 103.

You can use the **NOT** anywhere in these statements. **NOT** has priority before **AND**. It makes truth false and falsity true (in a sense).

Try putting **NOT** in front of the statement previously used—use the same values of X, Y, Z, and note your results: Where you got true (1) before, you should get false (Ø) now!

```
2Ø IF NOT ((X ≤ 4 OR X ≥ 8) AND (Y ≥ 4 OR
(Y = 2 AND Z < > 1))) THEN GO TO 5Ø
```

See result c in the table on p. 103.

Note: The inner parentheses are not really necessary, but they make things clearer and can't hurt as long as they balance out.

Also try it with this statement for better understanding of where some expressions have **NOT**s.

```
2Ø IF NOT X ≤ 4 OR X ≥ 8 AND NOT Y ≥ 4 OR NOT Y = 2 AND Z < >
1 THEN GO TO 5Ø
```

See result d in the table on p. 103.

LOGIC AND LET

Logic phrases can be combined with arithmetic operators in a **LET** statement such as

```
1Ø LET A = (X AND Y AND Z) * 1Ø
```

This statement would yield a value of 1Ø only if the logic phrase in parentheses is true—in this case all three variables must be true (not Ø).

Try values of Ø and 1 for X, Y, and Z above and add line 2Ø to see what happens.

```
2Ø PRINT A
```

Here **A** is either \emptyset or 1 \emptyset .

Try this statement:

```
1 $\emptyset$  LET A = (X > 4 OR Y < 3 AND Z <> 5) * 15
```

Here **A** will become 15 only under the specified conditions—both $Y < 3$ and $Z <> 5$ must be true, or $X > 4$.

The advantage of this trick is primarily one of saving space. We could have said:

```
1 $\emptyset$  IF X > 4 OR Y < 3 AND Z <> 5 THEN GO TO 2 $\emptyset$ 
2 $\emptyset$  LET A = 15
```

using more bytes! Sometimes a line like 2 \emptyset could get in the way, and other lines would have to be added to route around it.

Logical operators can be used to provide arithmetic values—for example, to adjust scores in games or alter results.

Rule: When you add a logical expression multiplied by a constant to another number, that other number is only affected if the logical expression is true.

Try this odd-or-even game to see how this can work.

```
1 $\emptyset$  PRINT "ODD(O) OR EVEN (E)?"
2 $\emptyset$  LET N = INT(RND * 1 $\emptyset\emptyset$ ) + 1
3 $\emptyset$  INPUT G$
4 $\emptyset$  IF CODE (G$) = 49 THEN GO TO 9 $\emptyset$ 
5 $\emptyset$  IF CODE (G$) = 52 - ((INT N/2) = N - N/2) * 1 $\emptyset$  THEN
    PRINT "CORRECT"
6 $\emptyset$  PRINT N
7 $\emptyset$  GO TO 1 $\emptyset$ 
9 $\emptyset$  LIST
```

In this program the computer establishes a random number between 1 and 1 $\emptyset\emptyset$ inclusive (line 2 \emptyset). You are to guess whether it is odd or even (line 3 \emptyset) by inputting "O" or "E". Line 5 \emptyset makes the comparison with the random number—if they match, it says correct. In any case, it prints the hidden number and lets you try again.

Note that your computer manual identifies code 52 for the letter "O" and 42 for "E". Line 5 \emptyset is a double logic statement. If the equality " $\text{INT}(N/2) = N - N/2$ " is true, then its value becomes 1 and the statement becomes:

```
IF CODE (G$) = 52 -  $\emptyset$  * 1 $\emptyset$  THEN . . .
```

and if the value were false, the statement would read:

IF CODE (G\$) = 52 - 1 * 10 THEN . . .

In the first case the value of your letter guess (odd or even) is compared with 42 and in the second case with 52.

Now note that **INT N/2 = N - N/2** can only be true for an even number! Hence the result for an even number is always 42 and therefore correct if you said "E" and wrong if you said "O"!

Can you understand the same logic when an odd number is used and how you get the "correct" response?

Lines 40 and 80 were added to save you grief. They allow you to get out of the infinite loop by depressing an "L". It will list the program so you can make corrections or changes.

LOGIC AND THE PRINT STATEMENT

Combining the logic operators with a **PRINT** statement can have interesting results also.

To understand, note the following chart:

<u>Logic statement</u>	<u>PRINT result</u>
A AND B	A if B is true 0 if B is false
A\$ AND B	A\$ if B is true Nothing if B is false
A OR B	A if B is false 1 if B is true

Now try a program to see it in action:

```

10 INPUT A$
15 CLS
20 IF A$ = "L" THEN GO TO 100
30 INPUT B$
40 PRINT (A$ AND A$ < = B$) + (B$ AND A$ > B$)
50 PRINT (A$ AND A$ > = B$) + (B$ AND A$ < B$)
95 GO TO 10
100 LIST

```

This program will arrange any two words in alphabetical order. Lines 40 and 50 compare A\$ with B\$. Line 40 will print either A\$ or B\$ depending upon which **AND** statement is true. Since both A\$ and B\$ exist, the left half is of

course true. According to the rules above, if the second-half value is true it will print the first half of an **AND** statement. (**AND** is the only possible operation in a logic **PRINT** statement using string variables.) Line 50 is written in reverse—"less than" rather than "greater than"—and will print the second word.

Lines 20 and 100 were added simply to allow you to get out of the loop without pulling the plug! Line 15 keeps the screen clean.

You can also put two words in alphabetical order using **IF** statements in place of lines 40 and 50:

```
40 IF A$ < B$ THEN GO TO 70
50 IF A$ > B$ THEN GO TO 80
70 PRINT A$, B$
75 GO TO 100
80 PRINT B$, A$
```

As you can see, five statements are required, whereas before only two were required. Counting bytes, we find: Lines 40, 50 were 52, whereas lines 40–80 are 79. (You can also combine **IF** and **PRINT**, as we did earlier, using only 38 bytes.)

Only your imagination limits your use of this Timex/Sinclair characteristic of handling strings and logic.

Let's try something with **OR**:

```
10 INPUT A
20 INPUT B
30 PRINT (A OR B > = 7) + (B OR A < > 3)
```

Try different values of A and B and see what results you get!

Line 30 reads as follows:

If "B is greater than or equal to 7" is false, **PRINT** A; and if it is true, **PRINT** 1 (the truth value). When "A is greater than or less than 3" is false, **PRINT** B; and when true, **PRINT** 1 for B.

Here's the results and analysis of sample tries:

<u>A</u>	<u>B</u>	<u>B > 7</u>	<u>A < > 3</u>	<u>Print</u>	<u>Result</u>
2	5	F	T	A + 1	3
4	8	T	T	1 + 1	2
3	5	F	F	A + B	8
3	8	T	F	1 + B	9

Only in the third sample was the result A + B or the actual sum of the original numbers. You can see that by using selected logic statements you can affect the outcome of an addition of two numbers. In the example the only time you will get the sum of A and B is when A is equal to 3 and B is less than 7. So if you want numbers to do strange (but predictable) things, such as in a game, you now have the tools!

SAMPLE PROGRAM

You are now prepared to write a simple arithmetic program using string and logic expressions.

Objective: Write a program which will display at random various math problems—limit it only to addition, subtraction, and multiplication. Allow the student three tries to get it correct. After every five problems print out the student's score.

Limitations:

No division problems.

No problems which require a negative result.

No numbers larger than 10.

Limit multiplication tables up to 5s.

Plan: Prepare a systems flowchart showing the flow of the information and process you want.

Step 1: List the processes in simple form:

1. Present the problem on the screen.
2. Allow the student to answer.
3. Give the student three chances.
4. Print the correct answer.
5. Keep a count of right and wrong answers.
6. Keep a count of the number of problems.
7. Post the final score.

Step 2: Post the steps on a chart (see Figure 5.1).

Note that several decisions must be made:

1. Is the answer given correct?
2. Has the student had three tries?
3. Did the student have five problems?

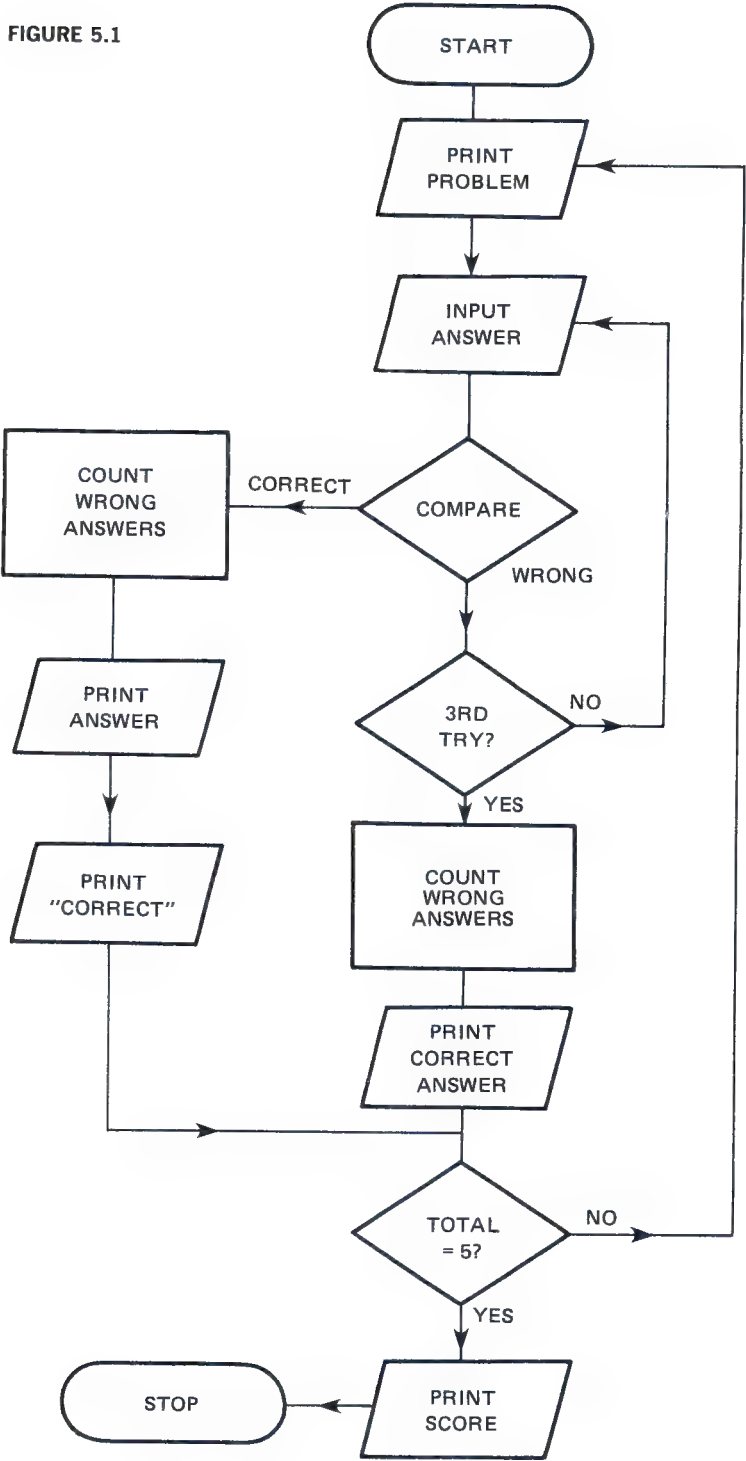
Step 3: Study your chart. The decisions, you note, can be accomplished by **IF** statements. The score can be kept by a counter. The loops are achieved with **GO TO** statements. Nothing especially complicated—except how do we obtain and post the problem automatically?

Step 4: Further detail the processes in determining the math problem:

1. Define the form as $X + Y = \text{answer}$.
2. Create the X number as a random number between 0 and 10.
3. Create a second number, also as a random number.
4. Create the math symbol (+, −, *) randomly.
5. Print out the whole problem, including the “equals” sign, realizing you must be able to get an answer from it.

Step 5: Create a flowchart of this detail, remembering that you need to make several decisions while creating the problem:

FIGURE 5.1



1. Is the problem going to give a negative answer—or, in a subtraction problem, is $X < Y$?
2. Is the problem a subtraction?
3. Is the problem a multiplication, and if so, is either number larger than 5?

If any of these conditions exists, you want to *not* print them and start again (see Figure 5.2).

Step 6: Rewrite your flowchart using BASIC terminology. But remember, we learned earlier that to print a problem it has to be a string variable; otherwise, the computer prints its equivalent. Also remember that you can “add” strings together in a special way. So in a problem like $X - Y =$, we want to present the X , $-$, Y , $=$ all as components of a string which can be converted to a numeric later for evaluation. Hence your steps shall include these as string variables.

Where we would have said `LET X = INT(RND * 11)` to obtain numbers 0 to 10, we now want to say `LET X$ = STR$ INT(RND * 11)`.

Doing the random sign function is a little tricky. First determine the code numbers for these functions either from your manual or by commanding `PRINT CODE “+”, CODE “-”, CODE “*”` and you will obtain 21, 22, 23. Therefore, you can call out the symbol by saying `CHR$L`, where L can be only 21, 22, 23.

In this manner you have created three string variables— $X\$$, $Y\$$, and `CHR$L`—and can now effectively combine them (see Figure 5.3).

Step 7: Now let’s write the program as we know it so far, following our flowchart (Figure 5.3) and assuming a correct answer:

```

30 LET X$ = STR$ INT (RND * 11)
35 LET Y$ = STR$ INT (RND * 11)
40 LET L = INT (RND * 3) + 21
45 IF VAL X $ < VAL Y$ AND L = 22 THEN GO TO 30
48 IF (VAL X$ > 5 OR VAL Y$ > 5) AND L = 23 THEN GO TO 30
50 LET P$ = X$ + CHR$ L + Y$
55 PRINT P$; “=?”
65 INPUT A
70 IF A = VAL P$ THEN GO TO 110
110 PRINT “CORRECT”
115 LET R = R + 1
120 PRINT P$; “=”; VAL P$
125 IF W + R = 5 THEN GO TO 140
130 GO TO 30
140 PRINT “YOU HAD #”; R; “# CORRECT” AND #”; W; “#
      WRONG”

```


FIGURE 5.2

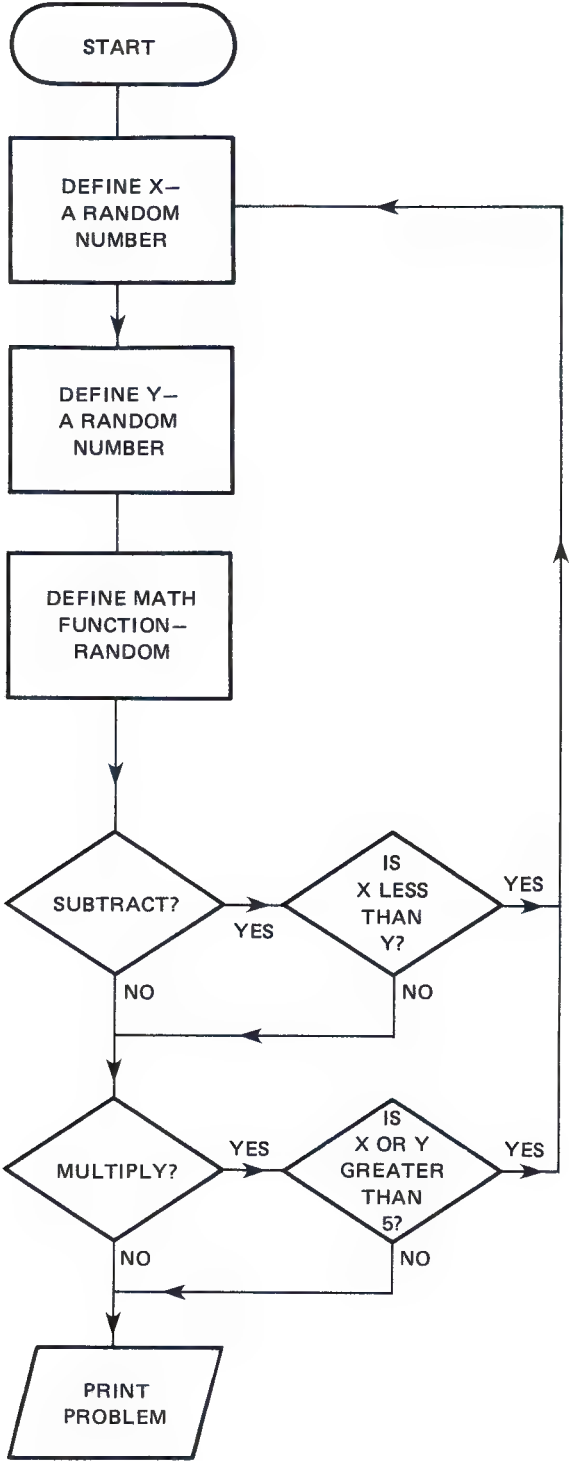
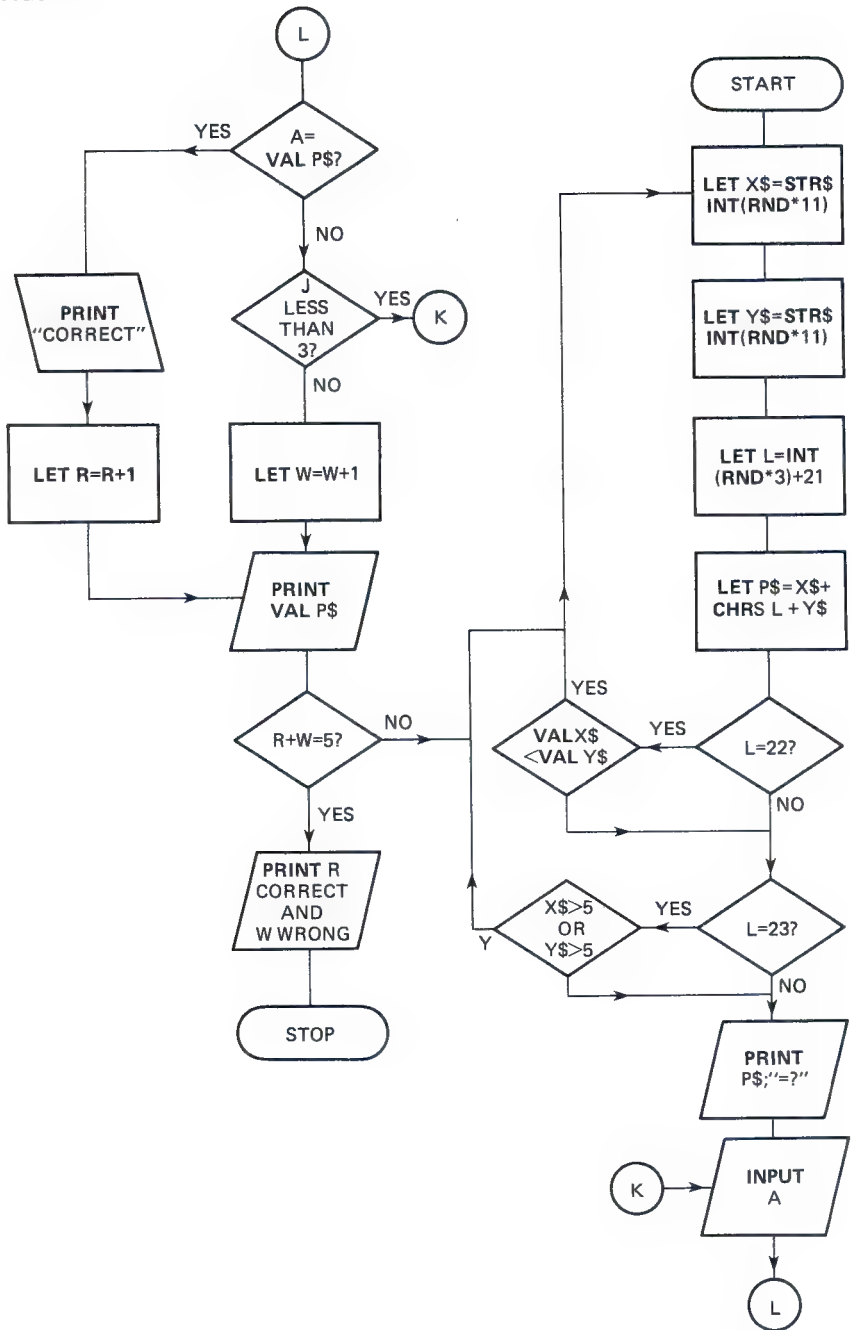


FIGURE 5.3



Step 8: **RUN** the program so far—it should work as long as you give correct answers. **OOPS**—error! You discovered an error on line 115 since R was not defined. So you add an initializer:

```
10 LET R = 0
```

RUN. There's an error on 125; so you add 15 **LET W = 0**.

Step 9: Go back and follow the route in your flowchart when you put in a wrong answer. Since line 70 gave a routine for a correct answer, the incorrect path can go after line 70:

```
80 IF J < 3 THEN GO TO 65
```

```
90 LET W = W + 1
```

Now **RUN** the program and you realize that J is not defined, for it only responds to correct answers; so you add an initializer and adder:

```
20 LET J = 0
```

```
75 LET J = J + 1
```

Now when you **RUN**, you discover after a while that you don't get more than one try to solve a problem. You study your program and realize the value of J keeps going up, and is never reset to zero because you only return to line 30.

You could change everything and **GO TO 20**, which is fine as long as the counter values of R and W are before it. But at this point it is easier to delete line 20 and add

```
60 LET J = 0
```

Step 10: You **LIST** your program and study it. Follow through the logic and compare it with a **RUN**.

You find it works OK when answers are all correct—but things go awry when wrong answers are made. It prints "correct" even when wrong!

You see that after line 90 is line 110, which is the routine for correct answers. You want to get to line 120, so you add:

```
95 GO TO 120
```

You also remember that **RAND** will truly randomize the **RND** function, so you add:

```
20 RAND
```

Step 11: You **RUN** your program and are *proud* that it works. You suddenly realize you can make all sorts of refinements—such as instructions tell-

ing the student to put in answers, to try again, etc. You could add choices such as what arithmetic functions you want, what limits are on the numbers, etc. You could include negative numbers or enlarge it to fancy problems. You could give it levels of difficulty.

Ah—but don't forget that after all this work you might want to *save* this program. To do so add:

```
5 REM "MATH"
```

and then turn on your tape recorder and command

```
SAVE "MATH"
```

For the record, here's your final program:

```
5 REM "MATH"
10 LET R = 0
15 LET W = 0
20 RAND
30 LET X$ = STR$ INT(RND * 11)
35 LET Y$ = STR$ INT(RND * 11)
40 LET L = INT(RND * 3) + 21
45 IF VAL X$ < VAL Y$ AND L = 22 THEN GO TO 30
48 IF (VAL X$ > 5 OR VAL Y$ > 5) AND L = 23 THEN GO TO 30
50 LET P$ = X$ + CHR$ L + Y$
55 PRINT P$; "="
60 LET J = 0
65 INPUT A
70 IF A = VAL P$ THEN GO TO 110
75 LET J = J + 1
80 IF J < 3 THEN GO TO 65
90 LET W = W + 1
95 GO TO 120
110 PRINT "CORRECT"
115 LET R = R + 1
120 PRINT P$; "="; VAL P$
125 IF W + R = 5 THEN GO TO 140
130 GO TO 30
140 PRINT "YOU HAD #"; R; "# CORRECT AND #"; W; "#
    WRONG"
```

SUMMARY REVIEW

In this level you have expanded your programming abilities into the realm of the string variable. You have learned to use the string functions **VAL**, **LEN**, **STR\$**, string dimensions (**DIM A\$**), and string slicing (**TO**). You have developed your use of the operands **<**, **>**, **<>**, **<=**, **>=**, **=**, and the logic functions **NOT**, **OR**, and **AND** as well as the **IF** statement in complicated expressions and conditions.

You have an understanding of the Timex's special features using logic in **LET** and **PRINT** statements.

You have also learned a practical application and have created a program you can use over and over.

In the next level you shall further develop your programming skills, as well as cover the balance of the keys not covered.

Now proceed to the exercises to clarify the points and enhance your abilities.

EXERCISES

5.1 Answer true or false: The result to

```
PRINT VAL(STR$ LEN "TODAY" + "4") is 9.
```

5.2 Determine the results of the following program:

```
5 LET A$ = "SMILE"
10 DIM B$ (5, 5)
15 FOR C = 1 TO 5
20 LET B$(C) = STR$ (4 * C)
25 NEXT C
30 PRINT VAL B$ (3)
35 PRINT LEN B$ (4)
40 PRINT B$ (3), B$ (4)
```

5.3 What is the result of the following?

```
PRINT "BEE" - "E"
```

5.4 What is the value of the following?

```
PRINT ACS (VAL(STR$ LEN "HI" + "-1.5")) * 180/PI; "#
DEGREES"
```

5.5 What is the difference in the results of these two programs?

1. 10 DIM A\$ (5,4)
15 LET A\$(3) = "SUPER"
20 PRINT A\$ (3)
2. 10 LET A\$ = SUPER
20 PRINT A\$ (3)

5.6 What is the result?

```
10 LET B$ = "WE LOVE TO GO TO THE ZOO"
20 LET B$ (4 TO 7) = "LIKE"
30 LET B$ (22 TO 27) = "MUSEUM"
40 PRINT B$
```

5.7 Write a program and set up an array of the 12 months of the year using a dimensioned string variable. Then have the sentence "THIS IS THE MONTH OF _____." print out for a particular month. Make sure the period is at the end of the word using the slicer!

5.8 Given a matrix of 4 columns with 10 words in each column and allowing for the days of the week, what would the dimension statement be? And what could D\$ (3, 2, 4 TO 6) be?

5.9 Which of these conditions is/are true when X is 0?:

- a. IF X = 5
- b. IF NOT X
- c. IF X
- d. IF X = 5

5.10 What happens when X is 5 in Exercise 5.9?

5.11 Given the following statement, complete the table:

20 IF NOT X < > 3 AND Y > 4 OR (Z = 0 AND X ≥ 4) THEN GO TO 50

RUN #	X	Y	Z	RESULT
A	<input type="checkbox"/>	5	2	T
B	4	4	<input type="checkbox"/>	T
C	3	<input type="checkbox"/>	2	T
D	1	1	1	<input type="checkbox"/>
E	4	1000	0	<input type="checkbox"/>

RUN it to prove your answers.

5.12 Write the following three ways: If A is 1 or 2, then print B\$.

5.13 What are the differences among the following?

A, A1, A\$, A(I), A\$(I), "A"

5.14 You have a program. The result of some variable **V** is a whole number between 1 and 11. Write the portion of the program such that for each possible **V** you branch to a different statement.

5.15 What will this statement do to "A" if $X = 2$, $X = 1$?

10 LET A = (X = 2) * 15

5.16 Which of these mean the same?

- a. IF $X > 0$ OR $X < 0$ THEN STOP
- b. IF $X = 0$ THEN STOP
- c. IF NOT X THEN STOP
- d. IF NOT $X = 0$ THEN STOP
- e. IF $X < > 0$ THEN STOP

5.17 Which of these are identical?

- a. IF $X - 15$ THEN STOP
- b. IF NOT $X - 15$ THEN STOP
- c. IF $X = 15$ THEN STOP

5.18 Modify the odd/even program to clear the screen (CLS) after each turn and to keep score of the number of correct answers.

5.19 In the Sample Program teaching arithmetic, combine lines 45 and 48 into one line.

5.20 Add line 145 to the Sample Program:

145 PRINT AT 20, 25; PEEK 16396 + 256 * PEEK 16397 - 16509

What happened?

5.21 You want to include division in your Sample Program. You also want to prevent division by zero, and you want problems which give integer answers. What lines will you add or change? Try it!

5.22 What does your Timex print if the values of M and N in the following are 3 and 2?

PRINT (M OR $N < > 2$) + (N OR NOT $M < > 3$)

5.23 What is the result of

PRINT (A\$ AND $C > = 4$) + (B\$ AND $C = 5$)

where

A\$ = "GO TO #".

B\$ = "JAIL"

and when C is a roll of dice equal to 5?

LEVEL SIX

PEEKING, POKING, AND GAMING

Congratulations! You have finally made it! You've learned all you need to know to write any program you might need. Hence, this level will be dedicated to developing your programming ability through a deeper understanding of the inner workings of the Z-80 processing unit.

OTHER KEYS

Have we covered the use of all the keys on our keyboard? Not quite. We did not discuss the *period* (on the comma key) or the *colon* (on the Z key). Rest assured—you missed nothing. These punctuation marks have no special powers other than their normal English usage. Hence you will only use these within quotes as string characters.

Well, what about the quote on the Q key? So far, when using quotes for strings we used the P key symbol both to open and close strings. But suppose you want a quote to appear in a string as a string? That will be the time to use the literal quote on the Q key. For example, suppose you want an instruction to read:

DEPRESS THE "ENTER" KEY

Your print statement would be:

```
PRINT "DEPRESS THE" "ENTER" " KEY"
```

where the string symbol is the single quote of the P key and the literal double quote is the Q key. Try the example.

Other keys not yet covered include the **GOSUB** and **RETURN** keys, which will be discussed in Level Seven. The **LPRINT**, **LLIST**, and **COPY** keys are strictly for use with a printer. Since you already understand **PRINT** and **LIST**, you know exactly what the **LPRINT** and **LLIST** commands will do—the only difference is that they will do it on paper instead of the TV screen. One of the major differences is that the printer is not limited by a “bottom” and hence will continue to print without the need for the **CONTINUE** command. Normally, you write the program as usual, then give the command **LLIST** to list the entire program for proofreading or giving to someone. To print the results, simply run the program and command **COPY** to get what is on the screen. In some instances you might want to use the **LPRINT** in a program which has final results requiring more than one screenful.

The only keys we haven't covered yet are the **PEEK** function and the **POKE** command on the 0 key, plus the **USR** function on the L key. **POKE** and **PEEK** are BASIC terms that allow you to speak to your computer directly without using machine language! (Machine language uses only numbers to direct the computer and bypasses the BASIC ROM chip. It is therefore more direct and quicker.)

We are able to talk to the computer normally using the BASIC you already know.

COMMUNICATING IN BASIC

Try the following exercise—an attempt to print a graph on the screen:

```
10 FOR N = 0 TO 63
20 PLOT N, 22 + 5 * TAN (N/32 * PI)
30 NEXT N
```

When **RUN**ning this program you can come up with an error. Why? And how do you correct it?

The error is a B. This means it is out of range—yet part of a curve is shown on the screen!

You can ask where on the curve the error exists. So you command:

```
PRINT N
```

You then get a 14—this is the next value of N. Apparently this results in a tangent value beyond the screen's range—it approaches infinity. So try values of N until a curve again appears on the screen.

```
LET N = 14      CONT
LET N = 15      CONT
```

Finally with $N = 17$ a portion of the curve appears. You now know a discontinuity exists with values of N of 14, 15, and 16.

However, you again have an error B.

```
PRINT N gives you 46
```

As above you discover a value of 49 will let it display a curve. So with:

```
LET N = 49      CONT
```

the curve continues and, if you have 1K RAM, fails on error 4—out of memory. A **CONT** command will display the balance to $N = 63$.

So how do you get your original program to print the entire curve? Well, you know the points of discontinuity are 14, 15, 16, 46, 47, and 48, so you include a statement to divert the program around these values. In BASIC this would be an **IF** statement with the **OR**:

```
15 IF (N > 13 AND N < 17) OR (N > 45 AND N < 49) THEN GO TO 30
```

Now **RUN** your program and see the results.

You can interact with your Timex anytime by commanding a printout and continuing from that point. There are times, however, when you need to access the memory banks directly and put something in it. This involves an intimate knowledge of the internal memory structure.

MEMORY STRUCTURES

You may recall from Level Three that 8 bits make a byte and byte makes a *character* or *token word*. The memory chip is actually constructed with 8-bit storage bins. Each byte has an address. Since they are all on the same “street,” each has a different “house” number. These numbers start with 0 and can go up to 65,535.

Why this magic number? Is this the limit? Well, 65,535 is equivalent to 64K, since in computer language 1K is 1024 bytes. But to understand the limitation we must understand the nature of the byte.

One byte has 8 bits each with a number 0 or 1 denoted by an electrical charge—akin to an on-off switch. This 8-bit byte represents decimal numbers in terms of base 2—binary numbers.

The decimal system we commonly use is a number system to the base 10, where only single digits are required to count up to 9. A base-8 system would en-

compass digits through 7 only. A base-16 system (common in computer use) uses 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. This *hexadecimal* (base-16) system is quite usable, requiring significantly *fewer* digits in the long run.

Here's how the *decimal* system is constructed. Each digit is a multiple of 10. The first (from right to left) digit is always in the 1's column, or 10 to the zero power. Hence 65,535 is the addition of the following where each digit is a place holder for ones, tens, hundreds, etc.

$$\begin{array}{r}
 6 \times 10^4 = 60,000 \\
 5 \times 10^3 = 5,000 \\
 5 \times 10^2 = 500 \\
 3 \times 10^1 = 30 \\
 5 \times 10^0 = \underline{5} \\
 65,535
 \end{array}$$

By the same method, the following number is also 65,535, based in powers of 2:

$$\begin{array}{r}
 111111111111111 \\
 1 \times 2^{15} = 32,768 \\
 1 \times 2^{14} = 16,384 \\
 1 \times 2^{13} = 8,192 \\
 1 \times 2^{12} = 4,096 \\
 1 \times 2^{11} = 2,048 \\
 1 \times 2^{10} = 1,024 \\
 1 \times 2^9 = 512 \\
 1 \times 2^8 = 256 \\
 1 \times 2^7 = 128 \\
 1 \times 2^6 = 64 \\
 1 \times 2^5 = 32 \\
 1 \times 2^4 = 16 \\
 1 \times 2^3 = 8 \\
 1 \times 2^2 = 4 \\
 1 \times 2^1 = 2 \\
 1 \times 2^0 = \underline{1} \\
 65,535
 \end{array}$$

You can see why we use the decimal system in everyday life! But the binary system is far more useful in a computer.

A zero in a position means that the power of the base is not included.

Hence

$$1\emptyset11 \text{ is } 1 + 2 + 8 = 11$$

where \emptyset represents 2^2 or 4 and was not included. Hence

$$1111 \text{ is } 1 + 2 + 4 + 8 = 15$$

The largest code you can store in 1 byte is one with all the 8 bits closed or

11111111

This number equates to 255. As you note from your handbook, that is the number of codes to represent all the characters and tokens of your Timex.

Your unit is constructed to allow addresses up to 2 bytes long. As you can see, 16 bits completely filled result in a decimal of 65,535. Hence this is the inherent limitation of your computer. How are these addresses used?

Well, the first 8K are devoted to the ROM chip—the translator—so all the machine language necessary is written here within 8192 locations. The subsequent 8K up to 16K are mirror images and presently not used. The first address usable in RAM is 16,384. With a 1K system the last address is 17,408. With a 2K Timex it is 18,432, and if you add the 16K, it becomes 32,768.

What about the other 32K? Well, these are available on restricted terms in the form of various memory packages on the market. Instructions are included in each package.

Certain addresses are used to store information, such as other addresses. The last byte available is called RAMTOP. Supposing you have 2K memory, the RAMTOP then is at 18,432. This number is stored at addresses 16,388 and 16,389 (the number is large enough to require 2 bytes).

The minimum system of 1K RAM has its RAMTOP at 17,408. Hence addresses 16,384 through 17,408 are available—but for what? We just noted that 16,388 and 16,389 are already used. In fact, for the BASIC ROM to work it needs to store various information. This is stored in RAM and uses addresses 16,384 through 16,508. How all of these are used is defined in your handbook. Hence only addresses 16,509 through 17,408 are really available to you—124 bytes less than you thought were there! And of these 900 bytes (or 1924 with 2K RAM), a further specialized separation exists.

Your memory space between 16,509 and 17,408 is distributed to seven sectors or “city blocks.” These include the program area, display sector, variables sector, work space, calculator stack, spare sector, and **GOSUB** stack.

The *program area* is used to actually store the instructions you put in. In an “empty” computer, zero bytes are used. In a “full” computer you will find approximately 600 bytes capacity here. The reason for this not being the full complement of 900 bytes is because the other sectors require some of the space.

The *display sector* utilizes the bytes to display on the screen what you see. The display is exactly 1 byte per character. The computer sets aside a minimum of 25 bytes for this purpose. A more complicated program which fills the screen consumes more bytes and can squeeze some from the program sector. You can depress **CLS**, clearing bytes in the display, and enter a longer program.

The *variables sector* stores the value of all variables being used. When you use a dimension statement, spaces for the array variables are stored in this sector. The computer sets aside initially 1 byte—this space stretches as it is needed.

The *work space* sector is a minimum 37 bytes and is used to enter or edit lines. This is the bottom two lines on the screen. The 37 bytes represent 32 bytes of character maximum per line plus 5 bytes for the line number. Of course, the display of the lines in this work space consumes bytes from the display sector.

The *calculator stack* sector normally has zero bytes, but expands to handle the calculations in a program when it is **RUN**. Data is stacked here and operated upon as required. Hence, it is possible to fill your computer and not be able to **RUN** because the calculator stack is squeezed to death!

The *spare sector* initially has the most bytes available—832. This is the area whence most is “borrowed” by the other sectors as it is needed. This sector is also the machine stack and is used by the Z80 processor chip to hold return addresses etc., and can therefore never be completely depleted.

Finally, the **GOSUB** stack of 4 bytes is used for storing routines. We will review this in Level Seven.

Your Timex maximizes its capability by having a flexible storage system. Each of these sectors is variable in length depending upon the need. You can see how it is impossible to say that a program of 6000 bytes will fit—it depends upon the program itself, its display requirements, quantity of variables, and amount of calculation needed. Hence, for simplicity, a claim of 2K means it will operate on your Timex with a RAM memory of 2K, although the program itself doesn’t really add up to 2K.

Remember when we determined that a **DIM**ension command over 159 spaces in 1K created an error? That was because the variable sector was squeezed to its limit. The additional 1K, however, was available in its entirety, as shown by the larger dimension capability!

In summary, the computer memory warehouse is segregated into sectors with only two fixed points—the beginning address 16,509 and the RAMTOP address 17,408 (1K). Within this area each sector has a beginning and ending. During the computer’s operation it needs to know where these sectors are located. In order to work on the display it needs to know where in the computer it started and where the last input was.

Hence, the Timex has an address book built into it—occupying the first 124 bytes of your RAM! These locations have their own fixed addresses filed in the permanent, nonchangeable ROM chip. Those locations which *store* the addresses of the beginning and ending of each sector are:

16396–7: Beginning of display End of the program

16400-1:	Beginning of the variables	End of the display
16404-5:	Beginning of the work space	End of the variables sector
16410-1:	Beginning of the calculator stack	End of the work space
16412-3:	Beginning of the spare/machine stack	End of the calculator stack
16386-7:	Beginning of subroutine stack	End of spare sector
16388-9:	RAMTOP location	

Obviously, this uses only 14 bytes of the 124 bytes; the balance are used for temporary storage of information and are not relevant for this study. Note that two address locations are needed, because they will store numbers up to 65,535, which requires 16 bits.

When 2K memory or the 16K Rampak is attached, the Timex will assess the total memory available and store the last byte into the RAMTOP location. The significance of RAMTOP is that you can move it for purposes of using space above it for machine language, isolating it from the BASIC programming and associated rules. The details of using this facility will be reserved for another manual. It encompasses the usage of the **USR** function.

PEEK-A-BOO

So now that we know where to go for these addresses, how do we access this information? You use the **PEEK** function. You simply command **PRINT PEEK address** and the computer will respond with a number 0 to 255. The addresses listed above require 2 bytes. Suppose you want to find the location of the beginning of the display file. You know this information is stored in addresses 16396 and 16397. However, since it is a 16-bit answer, to simply **PEEK** each address will not result in the correct answer. You would get instead two numbers each in the range 0 to 255. If, however, you changed these to binary code and strung out a 16-bit code and then converted the whole thing to a decimal, you would get the correct number. Another way would be to multiply the value of the second address by the value of a full byte or 256 like so:

PRINT PEEK 16396 + 256 * PEEK 16397

Try it—the answer is the location of the first byte of the display file. Since it is also the end of the program file, you can very easily determine the number of bytes in your program by subtracting the address of the first programming byte, which we know as 16509. You could put this in your program as the last statement—of course, the statement itself consumes bytes, too:

100 PRINT PEEK 16396 + 256 * PEEK 16397 - 16509

If you tried to determine the quantity of bytes in the display sector using

```
PRINT PEEK 16400 + 256 * PEEK 16401 - (PEEK 16396 + 256 *
PEEK 16397)
```

you would always get 25, because the display would first clear itself! So how do you determine the bytes in a display? Put the above command within the program as the last display item, and it will be displayed with the display itself!

To get a better understanding of the PEEK capability as well as the sector storage, RUN the following program. If you have 16K, run it with and without the pack connected.

The bytes in the program sector should agree exactly with what you calculate manually using the rules of Level Three.

```
5 LET M = 256
6 LET H = 16509
10 LET A = PEEK 16396 + M*PEEK 16397
20 LET B = PEEK 16400 + M*PEEK 16401
30 LET C = PEEK 16404 + M*PEEK 16405
40 LET D = PEEK 16410 + M*PEEK 16411
50 LET E = PEEK 16412 + M*PEEK 16413
60 LET F = PEEK 16386 + M*PEEK 16387
70 LET G = PEEK 16388 + M*PEEK 16389
90 PRINT "SECTOR/ADDRESS", "BYTES"
100 PRINT
110 PRINT "PROGRAM--"; H, A-H
120 PRINT "DISPLAY--"; A, B-A
130 PRINT "VARIABLE:"; B, C-B
140 PRINT "WORKAREA:"; C, D-C
150 PRINT "CALCS----"; D, E-D
160 PRINT "SPARES---"; E, F-E
170 PRINT "RAMTOP---"; G
CLEAR
RUN
```

Your results will be, with 1K RAM:

Program:	16509	474 bytes
Display:	16983	25
Variable:	17008	25
Work Area:	17033	6

```

Calculator:    17039      6
Spares:       17045     359
RAMTOP:       17408

```

With other than 1K RAM you will get a different RAMTOP and hence greater spare bytes, plus a larger work area.

The program sector is always exactly what you consume in the program in accordance with the rules defined in Level Three. Looking at this program, let's calculate bytes line by line:

<u>Line(s)</u>	<u>Bytes</u>
5	17
6	19
10 through 70: 35 each—	245
90	29
100	6
110 through 160: 23 each—	138
170	<u>20</u>
	474

The display sector is a minimum of 25 bytes—even though there is a display now; when line 10 was created there was no display!

The variable sector shows 25 bytes—at the time line 20 was computed, 25 bytes were used to store variables up to that point. This included 1 byte for the name and 5 bytes for the value. Hence for M, H, A, and B, 4 times 6 or 24 bytes were used. The computer has a minimum of 1 byte or 25 bytes total.

The work area in an empty computer is 37 bytes—a line requires 32 bytes. No line was being typed during the **RUN**, and hence only 6 bytes were left for work space—needed for a command.

The calculator stack is normally empty—6 bytes indicates it was storing some number result at the time it was **PEEK**ed.

The spare stack is what is available at the moment it was read.

To show you how these addresses change, do the following after you **RUN** the program and are looking at the display:

GO TO 10

What changed? The variable sector increased to 55 because once the program had been completely run, all 9 variables (5 bytes each or 45 bytes) were stored with all their names (1 byte each or 9 bytes) plus 1.

The working area and calculator stack had been reduced to 0 (zero) since the program is now completed.

The spares adjusts to the changes. RAMTOP, of course, doesn't change.

How do we get a number on the display sector, since it never seems to change? Whenever you ask for the end of the display stack, the result is always 25 more than at start, because the computer responds to a command by computing and cleaning the display, allowing more space in other sectors momentarily. This is why your unit flickers (in the **FAST** mode).

You can get a quantity on the display by putting in the **PEEK** at the right location:

```
180 PRINT PEEK 16400 + M*PEEK 16401-A
RUN
```

Add this to your program.

Of course, this statement adds to the program resulting in a new starting address for the display. Your result still shows 25 bytes; however, at the bottom you'll see the result to line 180 as 168 bytes. You can count all the characters on the screen.

You should now have a better understanding of how the computer works and how you can *look* at any address or location within the computer. Can we now put something in an address?

Yes—using the **POKE** command.

POKING AROUND

The **POKE** command enables you to input anything into your Timex directly. For example, you could change the location of RAMTOP or modify your program or change data in your variables or affect the display.

Why would you want to do any of these things? If you needed computer space for machine language routines, you would move the RAMTOP. If you were working with a “deck of cards,” you could eliminate cards by poking in a value of zero. Remember, an address can only hold 1 byte or a code number up to 255. Hence you can insert letters, numbers, symbols, and even commands.

But be careful of your poking. A **POKE** in fun can result in a program crash! This means you lose your program and must start over! Due to the variable nature of the storage mechanism of your Timex, a mere **POKE** could shift everything, and you might **POKE** where you had not intended to. For this reason **POKE** is often tied directly with the **PEEK** function.

Following is a short program displaying a “hole” on the screen. It allows you to **POKE** stuff on the display in just that location. But beware! **POKE** only single characters.

```
10 PRINT AT 10, 15; "## "; TAB 47; "## "; TAB 47;
   "## "
20 INPUT A$
```

```

30 IF A$ = " " THEN STOP
40 POKE PEEK 16 396 + PEEK 16 397*256 + 46, CODE A$
50 GO TO 20

```

The overhead bar is our symbol to tell you to type in the character while in Graphics **G** mode. Hence the “number” symbol (#) with the overhead bar means you are to type in the **SPACE** after you get in the **G** mode, i.e., after depressing **GRAPHICS**. This results in dark spaces. Note that the center space does not have a bar and is therefore typed in **L** mode as a regular space.

Note that line 40 **PEEKs** into the location for the start of the display sector. It then adds 46 to that location because the position we want to poke into is 46 spaces from the beginning. This does not necessarily mean we have used 46 spaces on the screen.

But let's find out what is in all those spaces up to 46. To do so we write a program to **PRINT** this information after **PEEKing** these spaces. We also want to do this with a full display so we do not disturb the first 46 bytes.

So we add to the above program the following lines:

```

100 PRINT AT 15, 0; "DISPLAY#";
110 FOR A = 1 TO 46
120 PRINT PEEK (PEEK 16396 + 256*PEEK 16397 + A); "#";
130 NEXT A

```

Line 100 starts our data below the program's display so we don't interfere with it. *Note:* It is necessary to include the semicolon to continue data thereafter.

Line 110 is the **FOR-NEXT** loop requesting data on the first 46 bytes.

Line 120 first determines the address at which the display sector actually begins. Then it adds “A” bytes to that. Then it **PEEKs** that new address to see what is stored there. Then it prints out the value of the byte in decimal.

To **RUN** this, first delete line 50; then **RUN**. When asked for a string variable, input the letter “Z”. Your results are:

```

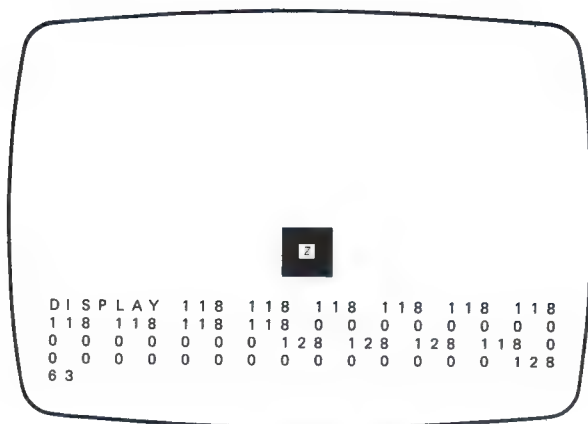
DISPLAY  118   118   118   118   118   118
118 118   118   118   0 0   0 0   0 0   0 0
0 0 0 0   0 0   0 1 2 8 1 2 8 1 2 8 1 1 8 0
0 0 0 0   0 0   0 0   0 0   0 0   0 0   128
63

```

These codes when converted by **CHR\$** (look it up in your manual or in Appendix C of this book) tell you that:

- The first 10 bytes say “ENTER”.
- The next 15 bytes say “SPACE”.
- The next 3 bytes are “black spaces”.
- The next is a byte saying “ENTER”.

FIGURE 6.1



- e. The next are 15 bytes saying "SPACE".
- f. Finally there is a "black space" followed by the letter "Z".

When we said in line 10 to **PRINT** at line 10 on the screen and column 15, the display sector stored this as 10 ENTERs to roll it down 10 times and 15 SPACES to move it over. Each line in display requires an ENTER whether it is used or not. After the three graphic spaces were displayed, the **TAB** instruction in line 10 forced it to go to another line, requiring an ENTER followed by sufficient spaces to get to the location for the next symbol—another graphic (black) space! Then, of course, is the code for the letter or number you have inputted.

You can see why 25 bytes are set aside for display in an empty computer—to allow for an ENTER for at least 22 lines.

We could have told the computer to display exactly what's in storage—try this by changing line 120 to read:

```
120 PRINT CHR$ PEEK (PEEK 16396 + 256 * PEEK 16397 + A);"##";
```

Note that the results are a string of question marks plus spaces plus the symbols used in creating the box plus the letter you put in. The question marks are for the ENTERs because they are stored as 1 byte.

Want to see what happens in your Timex with token words? And the existing program? What might you expect if you inject the following line? Try it:

```
5 PRINT AT 5, 4; "LETS STOP"
```

For **STOP** use the token word on A key.

Now **RUN** your program. Note that your letter Z which you input has now been shifted 9 spaces to the left out of your box. That is because the phrase

LETS STOP required 9 bytes in the display sector (although in the program sector it took only 5 bytes!). You still have 10 ENTERs, as before.

You can understand now why you really didn't have all those bytes available in writing a program. You can conserve bytes by careful display techniques as well.

What happens when a token word is forced into the display such as "STOP"? It obviously needs 4 bytes on the display. An injection of this in your little box on the display will result in a crash, because it pushes the entire display out of whack.

Try it! It only hurts those fingers because the program is lost!

It would also go berserk if you tried to **POKE** one of those bytes which have an **ENTER** in them.

So you can see, **PEEK**ing is relatively safe, but **POKE**ing can be disastrous if you don't know what you are doing. As you become more familiar with your computer, you will find more use for **POKE**. In any case, this introduction clarifies it enough for you so you understand what it is about next time you see it in a program. **POKE** is always followed by an address, a comma, and the code number (0 to 255) of what you want to inject. The address can be a complicated formula such as in the above program.

Enough said about the intricacies—let's do some fun programs. **RUN** this one for *fun*:

```

110 FOR T = 0 TO 60
120 LET A = T/30 * PI
130 LET SX = 31 + 9 * SIN A
140 LET SY = 21 + 9 * COS A
150 PLOT SX, SY
160 NEXT T
170 PLOT 28, 25
172 PLOT 34, 25
173 PLOT 31, 21
174 PLOT 32, 16
175 PLOT 30, 16
176 PLOT 33, 17
177 PLOT 29, 17
178 PLOT 31, 21

```

HANGMAN

We are all familiar with the Hangman game. A secret word is selected by an opponent and then we guess the letters. When wrong, a portion of a person is drawn on the gallows. When a correct guess is made, it is printed on the spaces

allocated for the secret word. If the person is completely drawn before the word is discovered, he is hung and you have lost!

If this program were put into a computer of sufficient size (16K RAM would do), you could play it directly, for it would pick out words from its own memorized vocabulary. If you only have 2K, you can have another person input the word when you are not looking.

How do we develop a program for our T/S 1000 or even the ZX81 with 1K RAM? With ingenuity and creativity! Although you might have 2K, it is to your benefit to learn to write the program in 1K because it sharpens your wits.

First, we think out the requirements for Hangman and determine just what we want the computer to do:

1. Display a gallows upon which to hang a person.
2. Have a secret word accepted and print out dashes under the gallows, one for each letter.
3. Provide for acceptance of guesses.
4. If a guess is correct, replace corresponding dash(es) with the correct letter(s).
5. If a guess is wrong, post the letter in error on the screen and add to the person being hung.

We now proceed with writing the BASIC to do the above:

1. We shall use graphics to display a gallows. We will locate it in the upper left of the screen (later you will see how this saves space). Our gallows will consist of a hook and top—the base is optional if you have 2K:

```
11 PRINT AT 1, 4; "̄3̄6̄6̄6̄6̄"
13 PRINT AT 2,6;"#̄8̄"
```

For a base you can add a loop and statement (2K RAM only):

```
12 FOR L=2 TO 8
13 PRINT AT L,8;"̄8̄"
14 PRINT AT 9, L + 2; "̄#"
15 NEXT L
```

Reminder: The bar above the character means depress the key when in **G** mode; the bar below means to **SHIFT** as well while in graphics **G** mode.

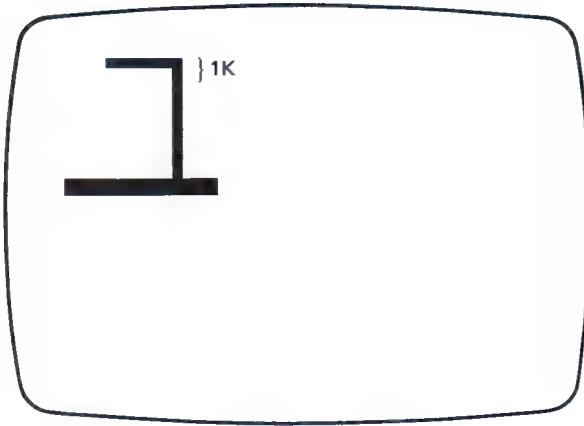
Line numbers were chosen for 1K users. For 2K you can use any number. Line 13 uses spaces to locate the gallows pole; however, it could have been

PRINT AT 2, 8; "8"

but we chose to use the space to keep the use of real numbers down, as will be seen later. It in no way detracts from the concepts, however.

You may **RUN** your program at this time and verify and construction of your gallows—it will look like this:

FIGURE 6.2



2. We are now ready to accept an unknown and print out dashes for each letter. First we request the secret word:

```
16 INPUT S$
```

If you have 2K, you could add the statement

```
PRINT "ENTER YOUR SECRET WORD"
```

and other fine points as we go along. The variable `S$` was chosen because we are working with a string.

To **PRINT** a dash for each letter of the secret word, we must determine the length of this word. We remember the string function **LEN**—so we incorporate it all in a loop like so:

```
17 FOR N=1 TO LEN S$
```

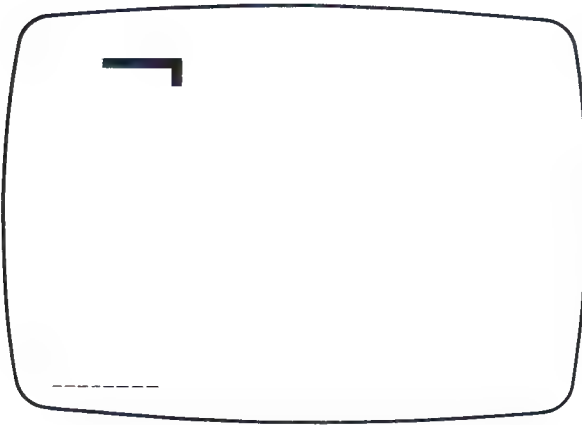
```
18 PRINT AT 20,N; "-"
```

```
19 NEXT N
```

We chose to print at line 20 to place it well under the gallows at the bottom left of the screen.

Now **RUN** the program as you have it thus far. **ENTER** the word "HANG-MAN" and watch the screen produce 7 dashes!

FIGURE 6.3



Once the word is hidden, the next person plays and tries to guess the missing letters.

We now need some means of accepting a single-letter guess and comparing it with each letter of the secret word. We do this with an **INPUT** and comparison loop:

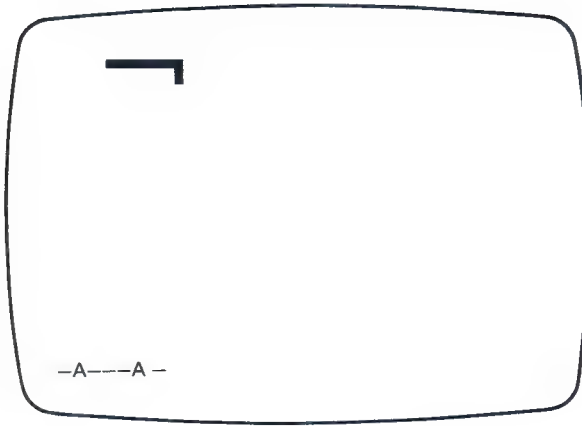
```
20 INPUT B$
50 FOR L=1 TO LEN S$
55 IF B$ <> S$(L) THEN GO TO 80
60 PRINT AT 20,L;B$
80 NEXT L
```

Line 50 scans each letter up to the full length. Line 55 uses a variable slicer of the string variable S\$ to designate each single letter. When it is not equal to the letter guess B\$, it becomes a true statement and one is directed to guess again. If line 55 is false, however, the computer operates line 60 and replaces the appropriate dash with the letter guess. It continues the cycle until the entire word has been scanned (for there could be another letter the same as your guess). Yes, by the way: Line 55 could have been written

```
IF NOT B$ = S$(L) THEN GO TO 80
```

Now **RUN** your program. **ENTER** the word HANGMAN. With 7 dashes, now enter the guess of letter A. What happened?

FIGURE 6.4



It replaced some of the dashes with the letter A. But it also stopped. You can see it scanned the whole word and inserted the letter wherever applicable.

So now we want the program to accept another letter guess until the word is completed. We simply redirect it to line 20—the acceptance of another letter string, like so:

```
85 GO TO 20
```

Caution, don't **RUN** yet, or you will get into an endless loop. We must also add some statements to get it out of the loop. Logically, you would want it out of the loop when all the dashes have been replaced by correct letters and the word is discovered.

Looking at the loop of lines 50, 55, 60, and 80 we see that we go to line 60 only when a correct letter is found. So let's include a counter, and when that counter is equal to the number of letters in the word we will leave the loop and **PRINT** something.

```
3 LET M = 0
70 LET M = M + 1
75 IF M = LEN S$ THEN GO TO 1200
1200 PRINT "YOU WON"
```

Line 3 is the initializer for the counter. Now **RUN** your program using **HANGMAN** as the secret word. Try guessing wrong letters—nothing happens. Once all letters are correctly "guessed," it printed "YOU WON" directly under the secret word.

So far, you have the beginnings of a Hangman game. You've developed a response for correct answers. Now how about wrong answers? The letters in error we would like to see on the screen also—say at the top, above the gallows.

Let's once again look at the loop of lines 50–80. What happens with a wrong guess? Line 55 says to go to line 80, i.e., repeat the loop. Once the whole word is analyzed, we are sent to line 20 again via line 85 for another guess. Somehow we need to avoid going back to line 20 until we acknowledge the wrong guess, yet we want to go to line 20 directly if we made a correct guess. So let's use a code. Insert line 65:

```
65 LET Q = 1
```

Since it is within the loop, Q will be the value 1 whenever a correct guess is made. We will make $Q = 0$ outside the loop like so:

```
45 LET Q = 0
```

Note that this value of Q is placed after the letter guess but before the analysis loop 50–80. How does this help us? We'll change line 85 to a conditional statement. That is, when Q is a 1 (true), meaning a correct guess is made, we can proceed to another guess:

```
85 IF Q THEN GO TO 20
```

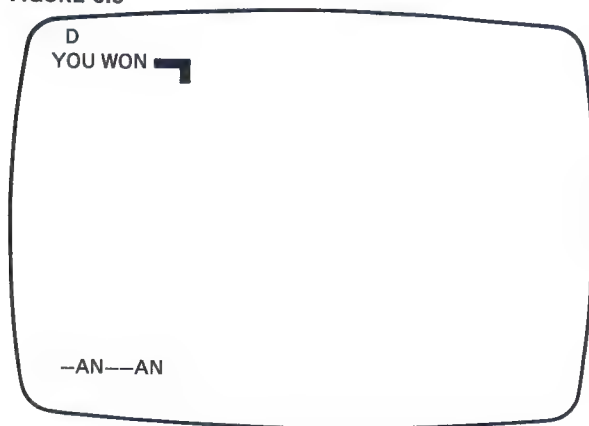
Hence if Q is still 0, it means that if after having gone through the loop for each letter of your secret word no correct letter was found, we can then proceed to the next line and do something with it such as **PRINT**:

```
95 PRINT AT 0, 1; B$
```

Now give it a practice **RUN** and observe what it does when you use "HANG-MAN" as the secret word and guess the following letters:

A N D

FIGURE 6.5



Well, this is not exactly what you want. Your first wrong guess stopped the program and the display says, "YOU WON!" It did, however, print the letter guessed.

Back to the drawing board!

Studying our program, we realize that after line 95 we don't wish to go to line 1200. We realize we need to allow for more than one wrong guess! But how many? Let's try 10 guesses.

Let's put in a line after line 95 which will send it back to line 20:

```
100 GO TO 20
```

Since this can go on indefinitely, we realize we better, add a counter, and allow it to only go to 10 and then get us out of the loop. So we add:

```
90 LET R = R + 1
99 IF R = 10 THEN GO TO 1100
1100 PRINT "TSK TSK"
```

Don't forget the initializer:

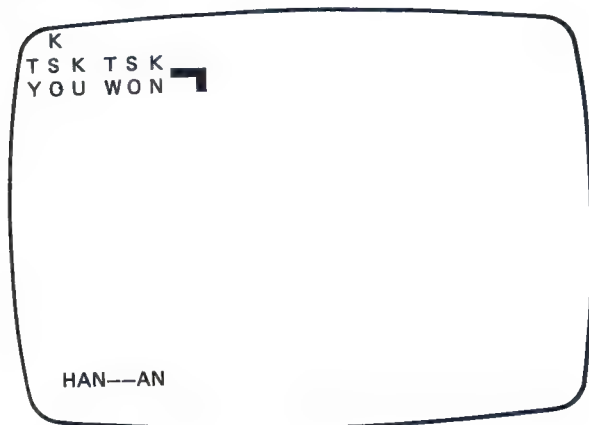
```
2 LET R = 0
```

Now try a **RUN** again. Guess the following letters:

A N D S T Y P V W F H J K

What happened?

FIGURE 6.6



Time to clean up our act again! What don't we like? Well, we don't display all the wrong letters—only the last one. And when we erred it still said, "YOU WON," and in any case it interferes with the gallows. Besides, there is no one being hung!

So now we tackle each problem, dealing first with the easiest. When we get to line 1100, we can prevent line 1200 and stop the program by inserting line 1150:

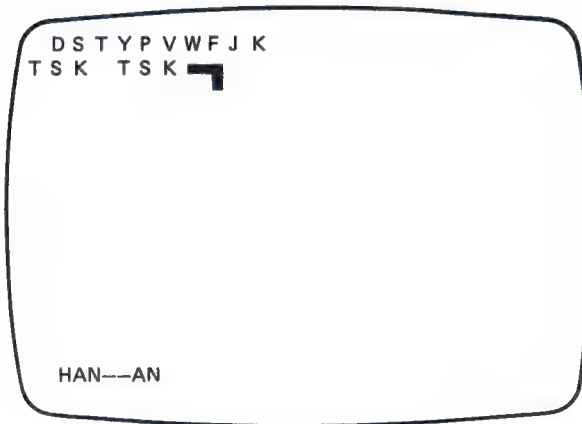
```
1150 STOP
```

If you are wondering why your listing on the screen is so short, it is because we are squeezing memory out of the video display sector (if you have only 1K RAM). Timex users will not have this problem unless you purposely moved your RAMTOP to 17408! We can print all the letters guessed wrong if we modify line 95 by using the counter's variable in the column location:

```
95 PRINT AT 0, R; B$
```

RUN your program again with the same guesses as before. What happened differently now?

FIGURE 6.7



Well it's getting better.

Now for the formidable task—getting the man hung on the gallows. This requires some thinking! First, let's define the completed man. He should have the

following parts—one head, one chest, two shoulders, two arms, one abdomen, one set of legs, and two feet—or 10 parts! Next, what graphics should we use? Most parts will be black: graphics space. The head should be different—we'll use the grey graphic on the **H** key. Hands shall be small—half-squares, as on the 7 key. The feet shall also be small, but to be different we'll choose the half-grey graphic on the **S** key.

Our objective will be to print out a portion of the person on the gallows for each letter guess which is wrong—and then to return to line 20 to continue guesses until the poor person is hung!

At this point, it is necessary to make a minor deviation, because some of our fellow readers have the ZX-81 with only 1K **RAM**. Instead of having *one* body part printed for each letter, we shall have *two* body parts printed for every *two* wrong guesses. If you have Timex 1000 with 2K **RAM**, follow along—you can always expand it. The technique is the same.

Hence, at every second wrong choice we want to print out two body parts and return to the program until we have printed the last part; then we want to go to line 1100.

Let's try the head and chest first:

```
200 PRINT AT 2, 4; "H"
```

```
250 PRINT AT 3, 4; "H"
```

If you are using 2K, you can also write it in this form:

```
200 PRINT AT 2,4; CHR$ 173
```

```
250 PRINT AT 2,4; CHR$ 128
```

To test this, change line 99 to read

```
99 IF R=10 THEN GO TO 200
```

RUN this as before using the same guesses:

```
A N D S T Y P V W F H J K
```

Well, at last we are beginning to achieve our goals! We printed two parts of our body. The trick will be to work from the error counter variable **R** in line 90 to get to the proper **PRINT** statement. We can do this by properly numbering the **PRINT** statements and using a **GO TO** as a function of **R** like so:

```
99 GO TO 100*R
```

and putting a return to line 20 after each pair of **PRINT** statements except the tenth, which will automatically lead you into line 1100:

```
300 GO TO 20
```

With this, the first error will send you to line 100, error 2 to 200, 3 to 300, and so on.

Let's continue with the addition of pairs of **PRINT** statements until the man is built. The two shoulders:

```
400 PRINT AT 3,3;"#"  
450 PRINT AT 3,5;"#"  
500 GO TO 20
```

the abdomen and legs:

```
600 PRINT AT 4,4;"#"  
650 PRINT AT 5,4;"#"  
700 GO TO 20
```

By now you might have discovered difficulty in editing errors because of lack of memory space (1K users). If so, simply move the arrow cursor to the line you are interested in. Then command **CLS**, clearing the screen. Depressing **EDIT** will now call for the line you wish to correct! Remember, there's always 25 bytes in the video sector.

The left hand and foot:

```
800 PRINT AT 4,2;"7"  
850 PRINT AT 6,3;"S"  
900 GO TO 20
```

Finally the right hand and foot:

```
1000 PRINT AT 4,6;"7"  
1050 PRINT AT 6,5;"S"
```

If you have 2K, you can **RUN** this program. However, with 1K you have a real problem. You might not have been able to even enter the last line. Delete it. Then depress **CLEAR**. This clears the variables sector momentarily so you can enter all the lines. Now what? Try **RUN**. **ENTER** the secret word "HANGMAN". So far, so good! Now continue with the guesses: A. And you get an error 4 on line 50, which means you ran out of space—probably in the variables sector because of the L in the loop.

So now 1K users have a program just squeezed in but insufficient memory to **RUN** it! Don't lose hope—we promised everything in this manual will work with 1K RAM as well. So we use our knowledge of memory consumption and ingenuity.

Remember, numbers require five extra bytes due to the floating decimal arithmetic capability. So an obvious solution would be to eliminate as many numbers as possible and replace them with letters!

First depress **CLEAR** to give us a little room to work in. Then **LIST**. Look at line 3. That zero costs us 5 bytes. Let's change line 3 to say

```
3 LET M=R
```

We have now saved those 5 bytes!

A common number in our program is 20—we use it in many **GO TO** statements. So add line 4

```
4 LET Z=20
```

So it costs us 15 bytes—but let's see how much it can save us to replace all 20s with the letter Z in lines 18, 60, 85, 100, 300, 500, 700, and 900, or 40 bytes, resulting in a net gain of 20 bytes! Go ahead—**EDIT** these lines.

Now try to **RUN**. Still an error. Well at least it got to the next line. More editing is still needed. You will notice that most numbers used are 6 or under. Let's establish some values:

```
5 LET F=1
6 LET A=2
7 LET B=3
8 LET C=4
9 LET D=5
10 LET E=6
```

However, each of these lines costs 15 bytes. Using letters we can further reduce these, like so:

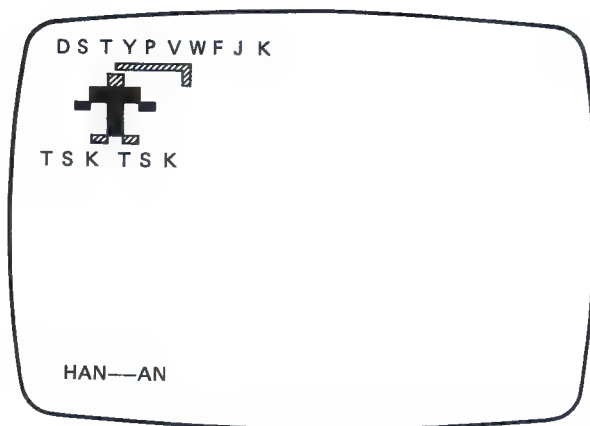
```
5 LET F=PI/PI
6 LET A=F+F
7 LET B=A+F
8 LET C=B+F
9 LET D=C+F
10 LET E=D+F
```

First **CLEAR**, then add statements 5 through 10. Now go through the entire program and replace numbers 1 through 6 with the above letters. Your program will then **RUN** even with long secret words. It should look like the following **HANGMAN** program. Note that line 1 was added to allow you to **SAVE** it on tape. Be sure to delete line 1 once loaded for maximum efficiency.

A good (bad) game would look like Figure 6.8.

```
1 REM "HANGMAN"
2 LET R = 0
3 LET M = R
4 LET Z = 20
5 LET F = PI/PI
```

FIGURE 6.8



```

6 LET A = F + F
7 LET B = A + F
8 LET C = B + F
9 LET D = C + F
10 LET E = D + F
11 PRINT AT F, C; "3 6 6 6 6 " (GRAPHICS—SHIFT 3 & 6)
13 PRINT AT A, E; "## 8 " (GRAPHICS—SHIFT 8)
16 INPUT S$
17 FOR N = F TO LEN S$
18 PRINT AT Z, N; "—"
19 NEXT N
20 INPUT B$
45 LET Q = 0
50 FOR L = F TO LEN S$
55 IF B$ < > S$ (L) THEN GO TO 80
60 PRINT AT Z, L; B$
65 LET Q = F
70 LET M = M + F
75 IF M = LEN S$ THEN GO TO 120
80 NEXT L
85 IF Q THEN GO TO Z
90 LET R = R + F
95 PRINT AT Q, R; B$

```

```

99 GO TO 100* R
100 GO TO Z
200 PRINT AT A, C; "H" (GRAPHICS—SHIFT H)
250 PRINT AT B, C; " #" (GRAPHICS—SPACE)
300 GO TO Z
400 PRINT AT B, B; " #"
450 PRINT AT B, D; " #"
500 GO TO Z
600 PRINT AT C, C; " #"
650 PRINT AT D, C; " #"
700 GO TO Z
800 PRINT AT C, A; "7" (GRAPHICS—SHIFT 7)
850 PRINT AT E, B; "S" (GRAPHICS—SHIFT S)
900 GO TO Z
1000 PRINT AT C, E; "7"
1050 PRINT AT E, D; "S"
1100 PRINT "TSK TSK"
1150 STOP
1200 PRINT "YOU WON"

```

Here are the features of your Hangman game:

1. There is essentially no limit to the size of secret word you might use—unless, of course, you can find one with over 30 letters!
2. You might wish to play guessing sentences, especially with little children. Once the sentence is in with spaces, you enter a space as the first guess, revealing the number of words on the screen and the number of letters in each word.
3. It will accept any character, including numbers and symbols.

Here some of the problems—and if you have 2K you can overcome these; in fact, it would make an excellent exercise:

1. You could use some instructions which tell the players what to input.
2. You could have it build body parts on a one-for-one basis.
3. You could prevent illegal guesses such as numbers or symbols.
4. Although entering a correct guess twice has no effect, it could think the word is complete when it isn't—you may want to prevent entering the same guess twice and even tell the player he did so!
5. You may wish to include an option to allow the player to choose to play another game and automatically **RUN** it.

You may also add variations to the game as your imagination dictates. Have fun!

The balance of this level will be devoted to presenting samples of games to stimulate your interest in further work.

NUMEROLOGY

According to students of numerology, the numeric value of your name represents your personality and your birth date represents your inner nature. Also important can be numbers representing the city you were born in and the city you now live in and important dates in your life.

In any case, you must first reduce the name into letters, where each letter has a numeric value, 1 through 9. These are then added together as digits until only a single digit is left, unless the number is an 11 or 22, for these have special significance. Dates are reduced by simply adding the digits until a single digit is achieved.

Here's the program to convert to single digits:

```

10 REM "NUMEROLOGY"
20 PRINT "FIRST NAME?"
25 INPUT F$
30 PRINT "M.I.?"
35 INPUT M$
40 PRINT "LAST NAME?"
45 INPUT L$
50 PRINT "BIRTHDAY?"
55 INPUT D$
60 PRINT "MONTH?"
65 INPUT A$
70 PRINT "YEAR?"
75 INPUT Y$
79 CLS
80 LET N$=F$+M$+L$
85 LET B$=A$+D$+Y$
100 LET Y=0
105 FOR I=1 TO LEN N$
110 LET Y=(CODE N$(I) - 37) + Y
120 NEXT I
130 IF Y=11 OR Y=22 OR Y<=9 THEN GO TO 160
140 LET L$=STR$Y
142 LET Y=0
145 FOR K=1 TO LEN L$

```

```

147 LET Y=VAL L$(K) + Y
150 NEXT K
155 GO TO 130
160 PRINT "YOUR NAME NUMBER IS#";Y
170 LET X=0
175 FOR J=1 TO LEN B$
180 LET X=VAL B$(J) + X
190 NEXT J
200 IF X=11 OR X=22 OR X<=9 THEN GO TO 240
210 LET B$=STR$ X
220 GO TO 170
240 PRINT "YOUR BIRTH NUMBER IS#"; X

```

Prior to each **RUN** on 1K depress **CLEAR**. Also delete line 10.

Here's what this program does: Lines 20 through 45 allow you to enter any letter string—specifically your name. If you don't have an entry, simply depress **ENTER**. Lines 50 through 70 allow you to enter numerical data—specifically, your birth date or really any date. Lines 80 and 85 add the component parts to produce a continuous string.

Lines 100 through 120 comprise a loop which converts the letters to numbers 1 through 9 and then adds them to one another. Note that it first converts to the code number.

Line 130 states that if the number is not reduced enough, continue with the number-reduction process. The K loop in lines 140 to 150 converts each digit to a numeric value and then sums them up. Just in case it still is reduced insufficiently, line 155 sends it to the test or recycles it. Line 160 prints the result.

Lines 170 through 220 do essentially the same thing lines 140 through 155 did—only this time for the date input.

How can you improve this? Look up some books on numerology. Have the program print out the meanings of the numbers. You will have a fun party game!

GUESS ME

Here's a simple game—the computer picks a number between 1 and 100. You try to guess the number. With each wrong guess it gives you a clue: too high or too low! It also keeps score of the number of guesses you took (up to 10).

```

1 REM "GUESS ME"
5 CLS
6 LET C=0
10 PRINT "I AM THINKING OF A NUMBER"

```

```

20 PRINT "BETWEEN ONE AND 100"
30 LET A=INT (RND*100)+1
35 PRINT
40 PRINT "WHAT NUMBER AM I THINKING OF?"
50 INPUT B
52 CLS
55 LET C=C+1
65 IF B=A THEN GO TO 100
70 PRINT ("TOO LOW" AND B<A)+("TOO HIGH" AND B>A)+(" #
   -SORRY" AND C=10)
80 IF C=10 THEN GO TO 150
90 GO TO 50
100 PRINT "YES, YOU ARE RIGHT"
110 PRINT
112 PRINT "I WAS THINKING OF #";A
115 PRINT
120 PRINT "IT TOOK YOU #";C;"# GUESSES"
130 GO TO 160
150 PRINT "I WAS THINKING OF #";A
155 PRINT
160 PRINT "IF YOU WISH TO PLAY AGAIN,","DEPRESS""Y"" "
165 INPUT A$
170 IF A$="Y" THEN GO TO 5

```

Note: Use the double-quote key for the double quotes around the letter Y in line 160.

LET'S DRAW

Here's an interesting program that lets you draw on the screen anything you want—using pixel dots. You can print your name in giant letters or draw pictures. With more memory you might want to add automatic circles and shapes. So here's your start:

```

5 REM "DRAWING"
10 LET PL=0
20 LET X=31
30 LET Y=21
50 IF INKEY$="5" AND X>0 THEN LET X=X-1

```

```

60 IF INKEY$="6" AND Y>0 THEN LET Y=Y-1
70 IF INKEY$="7" AND Y<43 THEN LET Y=Y+1
80 IF INKEY$="8" AND X<63 THEN LET X=X+1
90 IF INKEY$="E" THEN CLS
100 IF INKEY$="P" THEN LET PL=1-PL
120 PLOT X,Y
130 PAUSE 6200
135 IF PL=1 THEN UNPLOT X,Y
140 GO TO 50

```

Note: Add line 131 **POKE** 16437,255 if in **FAST** mode or on the ZX-80.

Lines 20 and 30 start your dot at location 31, 21. Lines 50 through 80 allow you to depress the arrow keys and direct your line movement. Line 90 allows you to erase everything except your new starting point—wherever you last were. Line 100 allows you to stop printing a line and move your dot from the end of one picture to the beginning of another. Depressing the letter P then activates line 135 to remove the mark. A subsequent press of P changes it back to the writing mode.

Have fun with this; it should give you lots of ideas. Line 130 can be altered to result in quicker motion responses. Try this program in both **SLOW** and **FAST** modes and note the difference. Experiment!

CATCH 'EM

This game tests your response time. It will display a person at the top of your screen. You are to determine where he is at by depressing the appropriate number key 0 to 9, representing positions from left to right. If you are correct, he throws his arms up and you score!

As an improvement, you might want to rearrange the positions on the screen from 0 1 2 3 4 5 6 7 8 9 to 1 2 3 4 5 6 7 8 9 0 to correspond with the arrangement of these keys on the keyboard.

```

1 REM "CATCH 'EM"
10 CLS
20 LET S=0
30 LET C=0
40 LET X=3*INT(RND*10)
50 PRINT AT 4,X;"#0#"
60 PRINT AT 5,X;"7 G 7"
70 PRINT AT 6,X;"#G#"
80 PRINT AT 7,X;"3 H 4"

```

```

90 PAUSE 90—C
100 IF INKEY$ <> STR$(X/3) THEN GO TO 180
110 PRINT AT 4,X;"Y O T"
120 PRINT AT 5,X;"# H #"
130 PRINT AT 6,X;"# H #"
140 PRINT AT 7,X;"T # Y"
150 LET S=S+1
160 PRINT AT 15,0;"HIT NUMBER#";S
170 PAUSE 180
180 CLS
190 IF C=20 THEN GO TO 220
200 LET C=C+1
210 GO TO 40
220 PRINT "YOUR FINAL SCORE IS#";S/C*100;"#PERCENT"

```

Note: If in **FAST** or on the ZX-80, add **POKE** 16437,255 to lines 91 and 171.

Line 90 controls the time allowed to find 'em! You could add levels of difficulty. Line 190 allows for 20 tries. Of course, you can change this, too.

SLOT MACHINE

Everyone wants to simulate Atlantic City or Las Vegas in one form or another. So here's a start. Go from here with more memory and add user-friendly instructions for the display. Using motion, you might make each section roll and so on.

This game displays three different symbols—when they match you have a win of five dollars. Losses cost you one dollar. Score is kept by keeping track of your winnings. The game is over if you run out of money or if you have won \$15. An improvement could be variable bets. In any case, **RUN** it!

```

10 REM "SLOT MACHINE"
20 PRINT "YOU HAVE $10 TO BET—GOOD LUCK!"
30 LET M=10
40 LET C=0
50 LET L=0
60 LET D=0
70 FOR I=1 TO 3
75 LET P=I*10-5
80 LET N=INT (RND*3)+1
90 GO TO N*100

```

```

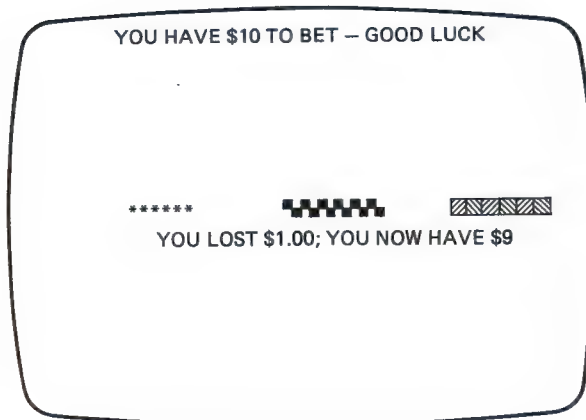
100 PRINT AT 10,P;"*****"
110 LET C=C+1
120 GO TO 400
200 PRINT AT 10,P;"Y Y Y Y Y Y"
210 LET L=L+1
220 GO TO 400
300 PRINT AT 10,P;"A H A H A H"
310 LET D=D+1
400 NEXT I
500 IF C=3 OR L=3 OR D=3 THEN GO TO 600
505 LET M=M-1
509 PRINT
510 PRINT "YOU LOST $1.00; YOU NOW HAVE $";M
520 PAUSE 240
530 IF M=0 OR M>=15 THEN STOP
535 CLS
540 GO TO 40
600 LET M=M+5
610 PRINT "YOU WON $5.00; YOU NOW HAVE $";M
650 GO TO 520

```

RUN in **SLOW** mode; otherwise, **POKE** 16437,255 on line 521 and delete line 10.

After line 535 you might wish to include the option to start over. Here's how the first run would look:

FIGURE 6.9



KNOCKOUT

Finally—the beginnings of a real game. In this one you try to shoot the enemy. When you achieve a hit you score a point. If you have the T/S 1000 or the ZX-81, run it in **SLOW** mode. With the ZX-80 8K ROM include the **POKE** 16437,255 statement in lines 31, 56, and 72. The **PAUSE**s used in this program should be changed to your personal liking. It will work fine as it is, however. Here's the program:

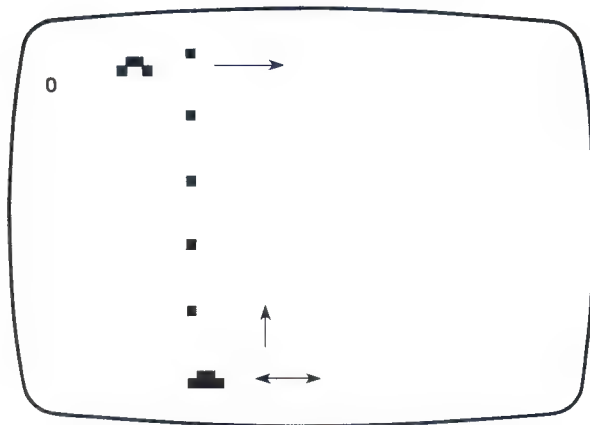
```
1 REM "KNOCKOUT"
5 LET V=2
6 LET K=0
7 LET R=5
8 LET M=9
10 PRINT AT 1,R; "T Y"
15 PRINT K
20 PRINT AT 21,M; "Q W"
30 PAUSE 52
40 IF INKEY$ < > "F" THEN GO TO 90
45 FOR Y=2 TO 42 STEP 8
50 PLOT M*2,Y
55 PAUSE 15
60 UNPLOT M*2,Y
65 NEXT Y
70 IF NOT (M >= R AND M <= R+1) THEN GO TO 90
75 PRINT AT 1,R; "$$$"
80 LET K=K+1
85 PAUSE 60
90 LET R=R+1
95 IF R>30 THEN LET R=R-30
100 LET M=M+(2 AND INKEY$="8")-(V AND INKEY$="5")
105 IF M>22 THEN LET V=0
110 CLS
115 GO TO 10
```

Line 10 displays the target. Line 90 moves the target, with line 30 giving the simulation of movement; and line 95 moves it behind the screen.

Line 20 displays the shooter, while line 100 allows you to control it by depressing the arrow keys. Line 105 is a glitch that prevents you from moving the shooter to the left once you have gone "too far" to the right. You could make this a random variable to make it more interesting!

Lines 40 through 85 allow for the firing of bullets; when you score a hit, line 70 allows for continuation to line 75—a display and addition to the score. Line 15 prints the score as you play.

FIGURE 6.10



By now you are reasonably proficient at programming with your computer. Hence there will be few exercises. At this point your best exercise is trying to improve on the programs in this level, and even creating your own programs!

The next level will be devoted solely to developing your program capabilities. Instead of gaming we shall look at some practical home business applications.

EXERCISES

- 6.1 Computer science specialists are presently developing a 16-bit processor. Can you calculate the number of possible addresses if you were limited to 2 bytes, where each byte is 16 bits long? Can you imagine the effects on the next generation of computers?
- 6.2 If you know how to calculate binary code, write a program to change from decimal to binary. If not, **RUN** the program listed in the answer section.
- 6.3 RAMTOP location is normally fixed in the computer automatically. However, you want to find out where it is and then relocate it by 40 bytes. What commands will you input to your Timex?
- 6.4 Given the following program, how many bytes are consumed in both the program and display sectors?

```
10 PRINT AT 19, 10; "MERRY CHRISTMAS AND HAPPY NEW  
YEAR"
```

Note: **AND** is the token word.

LEVEL SEVEN

PRACTICAL APPLICATIONS

Welcome to the world of computers! We can honestly say this now that you have achieved Level Seven. What you do from here on is up to you!

At this level we are interested in developing the practical side of computing. “After all,” your friends and relatives will say, “what do you do with it besides play games?” And you will show them!

At this level special exercises and the usual approach are no longer necessary. By now you should be self-encouraged to “exercise” the programs presented into more developed and sophisticated programs and even to go beyond that and create your own thing!

We will develop four types of programs—interest calculations, income tax work, data sort, and lottery selection.

GOSUB RETURN

First, however, we must not forget to introduce Mr. and Mrs. **GOSUB RETURN!** These two keywords haven’t really been necessary, but at times could really make life easier. Here’s how **GOSUB** looks in a program:

```
10 GOSUB 10000
```

It always refers to a line number. The number could be a variable, for example:

```
10 GOSUB A
```

or any expression which will result in a line number.

In a program **GOSUB 1000** really means: "OK, take a break from this program. Go to a subroutine starting at line 1000. Do that and then come back into this program." A *subroutine* is a lower-level program.

Once you have left your program and gone to a subprogram you will need a cue that says. "OK, this subprogram is finished—please **RETURN** to the main program and **CONTINUE** with the original program." This is all encompassed in one command:

```
1100 RETURN
```

Every **GOSUB** must have a matching **RETURN** at the end of the subroutine. Every **RETURN** statement must have a **GOSUB** somewhere. Of course, the **RETURN**s can be shared with many **GOSUB**s going to the same subroutine though not necessarily at the same location. For example, suppose you have written a program lines 1000 through 1200 (10 apart) with a **RETURN** statement at the end. You write a second program which needs to use the first program several times. You can very well have several **GOSUB**s such as **GOSUB 1000**, **GOSUB 1100**, etc. as long as each ends with 1200 **RETURN**!

You will see a good example of using **GOSUB** in the tax program we develop later.

THE LOTTERY NUMBER

Here's a program which all will find interesting—except probably the programmer! It is not at all difficult, but is rather straightforward. It is more oriented to the user. First it will ask whether a number is needed, and then for which lottery. You know, of course, that a number is easily generated at random using the **RAND** command and **RND** function. It will amaze those avid lottery players!

```
1 REM "LOTTERY NUMBER"
10 PRINT "NEED A LOTTERY NUMBER TODAY?"
15 INPUT A$
16 CLS
20 PRINT "WHICH GAME?"
25 PRINT
30 PRINT "A)#PENNA LOTTO","B)#N.J.PIC-6",
    "C)#PICK FOUR","D)#PIC 3 STANDARD"
```

```

35 INPUT B$
40 GO TO (CODE B$-32)*10
60 LET N=6
61 LET L=40
62 LET A=1
65 GO TO 100
70 LET N=6
71 LET L=36
72 LET A=1
75 GO TO 100
80 LET N=4
81 LET L=10
82 LET A=0
85 GO TO 100
90 LET N=3
91 LET L=10
92 LET A=0
100 RAND
105 CLS
110 FOR I=1 TO N
120 LET X=INT(RND*L)+A
130 PRINT X;"# #";
140 NEXT I
145 PRINT
146 PRINT
150 PRINT "ANOTHER NUMBER?"
155 INPUT K$
160 IF CODE K$=62 THEN GO TO 20

```

Now let's analyze it so you fully understand how it was written. Line 15 will accept any response—this just makes the user feel like there is some control. Line 30 lists four options A, B, C and D—most lotteries are three or four digits—selections D and C. Some states like Pennsylvania and New Jersey have special lotteries requiring a selection of six numbers—options A and B. There is a difference, however: Pennsylvania has a restricted selection of 1 to 40, whereas New Jersey has 1 to 36.

Line 35 allows the user to select a letter corresponding to his or her choice. Lines 110 to 140 comprise the loop which generates the numbers. It is used with all possibilities by using variables N, L, and A. L defines the range of possible

numbers—10 when using digits (0 to 9). A is used to prevent a selection of 0 plus include the value of L. N is the number of digits or selections needed.

Lines 60 through 92 assign values to these variables as required. Line 40 is the trick to get to the right variables. It converts the selection letters to a number and reduces it by 32 so that choice A results in $38 - 32$ or 6. When this is multiplied by 10 you get line 60!

Line 160 allows the user to go for another lottery number. It could have been written

```
IF K$="Y" THEN GO TO 20
```

However, if the user decided to respond to line 150 by writing YES, the computer could not respond. Line 160 as it is only evaluates the *first* letter—of course, we are in a pickle if the user responds with SI or JA or OK!

If your state has other lotteries or different rules, add these to your program.

Let's now go on to something a little more serious.

SORTING NUMBERS AND WORDS

A common use of computers is the reorganization of data, often in some kind of order—numerical or alphabetical. Following are two programs; one sorts numbers, the other sorts words. You will note that the two programs are virtually identical except one uses a numeric variable and the other uses a string variable.

What happens when you try to use the string variable for organizing numbers? After all, **INPUT A\$** will accept numbers as well!

First try these programs—then answer the question.

```
1 REM "SORT NUMBERS"
10 PRINT "HOW MANY?"
15 INPUT N
16 CLS
20 DIM A(N)
30 FOR M=1 TO N
35 INPUT A(M)
40 PRINT A(M),
50 NEXT M
55 CLS
60 FOR I=1 TO N
70 FOR J=1 TO N
80 IF A(I) >= A(J) THEN GO TO 120
```

```

90 LET B=A(I)
100 LET A(I)=A(J)
110 LET A(J)=B
120 NEXT J
130 NEXT I
140 FOR M=1 TO N
150 PRINT A(M)
160 NEXT M

```

```

1 REM "SORT WORDS"
10 PRINT "HOW MANY?"
15 INPUT N
16 CLS
20 DIM A$(N,10)
30 FOR M=1 TO N
35 INPUT A$(M)
40 PRINT A$(M),
50 NEXT M
55 CLS
60 FOR I=1 TO N
70 FOR J=1 TO N
80 IF A$(I) >= A$(J) THEN GO TO 120
90 LET B$=A$(I)
100 LET A$(I)=A$(J)
110 LET A$(J)=B$
120 NEXT J
130 NEXT I
140 FOR M=1 TO N
150 PRINT A$(M)
160 NEXT M

```

These programs are designed by virtue of line 80 to sort numbers from smallest to largest and words from A to Z. Change the symbol to \leq and the reverse effect will occur!

Lines 15 through 30 need to know how many quantities need to be sorted. Lines 30 to 50 comprise the input loop, during which an entire list is defined in any order and printed for information.

Lines 60 to 130 are a pair of nested loops which compare two items and switch locations if not in order. The variable B is used as the holding variable for

switching. The loops cause a comparison between the first number or word with each one thereafter, then the second item continues with a similar process. This technique forces ultimately the smallest or first to the top (known as the *bubble sort*). It is the simplest technique, but the most time-consuming. You might wish to research other techniques.

Numbers 1, 16, 11, 112, 1530 would be sorted out as 1, 11, 16, 112, 1530 with the number sort but as 1, 11, 112, 1530, 16 with the word (alphanumeric) sort. The latter looks at each digit only and in sequence; not the value of the whole string!

As an exercise the sort program can be modified to accept any number of items by dimensioning for maximum capability, allowing you to signal when you have reached the end of a list, letting M in the loop become N.

Now for some programs to help you make financial decisions.

INVESTMENT RETURN

Here's a program to determine the value of an investment. The actual program is simple—its value is in being “user-friendly” by providing proper cues and giving complete answers.

When considering an investment you have an idea of the amount you wish to invest, how long you can tie up your money, and what the interest rate might be. You are interested in how much you can earn for that investment.

```

1 REM "INVESTMENT RETURN"
10 PRINT "HOW MUCH ARE YOU INVESTING?"
20 INPUT P
25 CLS
30 PRINT "AT WHAT PERCENT INTEREST RATE?"
40 INPUT R
45 CLS
50 PRINT "FOR HOW MANY MONTHS?"
60 INPUT M
65 CLS
70 LET I=R/100
80 LET N=M/12
90 LET S=P*(1+I)**N
100 LET S=INT(S*100+.5)/100
120 PRINT "WITH AN INVESTMENT OF $";P;"#FOR#"; M;"#
    MONTHS AT AN INTEREST RATE OF #";R;"#PERCENT"
130 PRINT

```

```

140 PRINT "YOU WILL RECOVER $";S
145 PRINT
150 PRINT "INTEREST EARNED: $";S-P

```

Line 70 converts the percent rate to a decimal for computation. Line 80 converts the time period to years. Line 90 is the formula which relates principal, interest rate, time, and final sum. Line 100 rounds off the dollar value to the nearest cent.

Following is another "interest" program you might find useful:

```

1 REM "PURCHASE FINANCE PLAN"
10 PRINT "HOW MUCH IS THE PRODUCT?"
15 INPUT P
20 CLS
25 PRINT "BUYING ON TIME, HOW MANY MONTHS DO YOU HAVE
   TO PAY IT BACK?"
30 INPUT T
35 CLS
40 PRINT "WHAT IS THE ANNUAL INTEREST RATE?"
45 INPUT R
50 CLS
55 LET R=R/100
60 LET I=INT(P*R*T/12*100+.5)/100
65 LET M=INT((P+I)/T*100+.5)/100
70 PRINT "THIS PURCHASE WILL ACTUALLY COST YOU $";I+P
75 PRINT
80 PRINT "YOU WILL BE PAYING $";M;"# EACH MONTH FOR#";T;
   "# MONTHS OF WHICH $";I/T;"# IS INTEREST"

```

Often major purchases are offered on a finance plan wherein the interest calculated is attached to the original purchase price, creating a loan equal to the price plus interest. You then make a monthly payment a portion of which is an equal part of the interest charge. This program then advises you of your monthly payment and the portion which is interest.

MORTGAGE LOANS

Today a more common loan is the type which charges interest on the basis of unpaid principal. It is common with mortgages and other collateral loans.

The following program calculates the monthly payment which would pay off the loan in a required time period at a fixed interest rate. Try this program,

varying the interest rate, and you will realize why a change in rate can affect the entire economy.

```

1 REM "MONTHLY PAYMENT"
10 PRINT "HOW MUCH OF A LOAN OR MORTGAGE?"
20 INPUT P
25 CLS
30 PRINT "FOR HOW MANY YEARS?"
40 INPUT N
45 CLS
50 PRINT "AT WHAT INTEREST RATE?"
60 INPUT R
65 LET I=R/100
70 LET Q=(P*I*(1+I)**N)/((1+I)**N-1)
80 LET Z=INT((Q/12)*100+.5)/100
90 CLS
100 PRINT "A LOAN OF $";P;"# FOR #";N;"# YEARS AT #";R;"#
    PERCENT WILL COST YOU","#";Z;"# PER MONTH"
110 PRINT
120 PRINT "OR $";Z*N*12

```

Given the following inputs:

Line 20: 50000 Line 40: 25 Line 60: 14

your result will look like this:

FIGURE 7.1

A LOAN OF \$50000 FOR 25 YEARS AT
14 PER CENT WILL COST YOU
\$606.24 PER MONTH
OR \$181872

Many people will find it interesting to see what it would cost them to buy a house or co-op today!

Line 70 is the formula to determine the periodic payment for a time of N periods—in this case, the annual payment. Line 80 rounds it off and reduces it to a monthly value.

If you have 16K RAM, you can combine all these investment-type problems into one. Each could be a subprogram to a selection process which determines the nature of the problem.

INCOME AVERAGING

An even more practical and promising use of your home computer is for your income tax files. Since we have designed this manual for 1K, an appropriate tax program would be one evolving around Schedule G—Income Averaging.

Federal income tax Form 1040—Schedule G is divided into two parts: The first determines eligibility and second determines tax.

The program titled "INC AVE" defines eligibility. The taxpayer is required to provide income information (as determined from line 34 of the 1040) over the previous four years plus the number of exemptions taken in each year to determine each year's taxable income.

The following assumptions were made: (1) no income was earned outside the United States and excluded under sections 911 and 931; (2) no income was subject to penalty under 72(m)(5); and (3) no excess community income is included.

```

1 REM "INC AVE"
10 LET TT=0
20 DIM T(4)
30 DIM E(4)
40 PRINT "TAX YEAR AND TAXABLE INC?"
45 INPUT Y
50 INPUT TY
60 LET D=1000
62 CLS
65 FOR I=1 TO 4
70 PRINT "LINE # $ INCOME AND EXEMPTS TAKEN?"
75 PRINT
80 PRINT "YEAR#";Y-1
85 INPUT T(I)
90 INPUT E(I)
100 IF Y-I=1978 THEN LET D=750

```

```

105 IF Y-I>=1981 THEN LET E(I)=0
110 LET T(I)=T(I)-E(I)*D
120 LET TT=T(I)+TT
130 CLS
140 NEXT I
150 LET L12=.3*TT
160 IF TY-L12>30000 THEN GO TO 200
170 PRINT "SORRY, NOT QUALIFIED-USE TAX TABLES"
190 STOP
200 PRINT "YOU QUALIFY"
201 PRINT
205 PRINT "FOR TAX COMPUTATION LET"
210 PRINT "L14 BE#";L12
215 LET L16=.2*(TY-L12)+L12
220 PRINT "L16 BE#";L16

```

Lines 20 and 30 establish locations in the T/S 1000 to store information on the four years previous to this taxable year. Line 45 defines the tax year as stated on your tax forms. Line 50 accepts the taxable income as now defined directly on line 34 of your 1040. Line 60 is the value of exemptions after 1978.

Lines 65 through 140 form the loop which accepts previous years' data, makes them all equivalent, and adds them up using the initializer of line 10.

Line 100 reflects the \$750 value per exemption prior to 1979. Line 105 was inserted to cancel the effect of the exemption in the calculation of line 110 for years after 1980 since it already was considered in line 34 (1040). After tax year 1982 line 100 can be deleted, and after 1984 lines 105, 90, 60, and 110 could be omitted.

Line 150 calculates the 30 percent value of the total income over four years. Line 160 compares that with the most recent taxable income. If the difference is not greater than \$30000, line 170 is printed; otherwise qualification is determined and the values needed for the tax computation are displayed—lines 210 and 220.

Now we come to the second program titled "USING SCHEDULE Y." This program requests the values provided by the first. The only assumptions which affect the program itself are that there is no excess community income and no amounts received under penalty of 72(m)(5).

Since there are five steps which require tax computations, this is a good example of needing the **GOSUB** routine. Given the assumptions above, however, only two computations are necessary. In any case, the length of the computation portion justifies the use of the **GOSUB**.

Schedule G requires the use of the XYZ tax schedules. For simplicity and teaching purpose only, schedule Y for marrieds filing jointly was used as published

for the 1981 tax year. Due to the limitations of 1K RAM, not all of the Y schedule was included. If you have 2K, add lines 17000, 18000, 19000, 20000, 21000, 22000, 23000, and 24000. If you have 16K RAM, you can also include Tables X and Z plus a selection step.

```

2 REM "USING SCHEDULE Y"
250 PRINT "L14 AND L16?"
260 INPUT L12
280 INPUT T
290 CLS
300 GOSUB 10000
310 LET TX2=TX
320 LET T=L12
330 GOSUB 10000
340 LET TX2=TX
350 LET TAX=.9875*(5*TX1-4*TX2)
400 PRINT "TAX:#";INT(TAX*100+.5)/100
10000 IF T>3400 AND T<=5500 THEN LET TX=.14*(T-3400)
11000 IF T>5500 AND T<=7600 THEN LET TX=.16*(T-5500)+
294
12000 IF T>7600 AND T<=11900 THEN LET TX=.18*(T-7600)+
630
13000 IF T>11900 AND T<=16000 THEN LET TX=.21*(T-11900)
+1404
14000 IF T>16000 AND T<=20200 THEN LET TX=.24*(T-16000)
+2265
15000 IF T>20200 AND T<=24600 THEN LET TX=.28*(T-20200)
+3273
16000 IF T>24600 AND T<=29900 THEN LET TX=.32*(T-24600)
+4505
25000 RETURN

```

This will work in most cases. Remember, T represents taxable income, not gross income! In any case, the addition of the following will encompass all possible income levels:

FOR>=2K RAM

```

17000 IF T>29900 AND T<=35200 THEN LET TX=.37*(T-29900)
+6201
18000 IF T>35200 AND T<=45800 THEN LET TX=.43*(T-35200)
+8162

```

```

1900 IF T > 45800 AND T <= 60000 THEN LET TX=.49*(T-45800)
      +12720
2000 IF T > 60000 AND T <= 85600 THEN LET TX=.54*(T-60000)
      +19678
2100 IF T > 85600 AND T <= 109400 THEN LET TX=.59*(T-85600)
      +33502
2200 IF T > 109400 AND T <= 162400 THEN LET TX=.64*(T-
      109400)+47544
2300 IF T > 162400 AND T <= 215400 THEN LET TX=.68*(T-
      162400)+81464
2400 IF T > 215400 THEN LET TX=.70*(T-215400)+117504

```

Note that lines 300 and 330 send you to subroutine program which calculates a tax value TX for a given T value. Line 2500 is the **RETURN** command which sends you back to line 300 plus 1 (301) or 331 depending upon where you came from.

Line 300 calculates the tax on T which was established in line 280 as L16 value. Line 330 calculates tax on T, which is defined in line 320.

The value of TX calculated is set aside as TX1 and TX2 in lines 310 and 340. Line 350 does the actual calculation of the tax based upon income averaging in simplified form. The .9875 value is based upon the 1981 1¼ percent tax credit and may need adjustment for years after 1981.

You can see from lines 1000 through 2500 the advantage of using **GOSUB** and **RETURN**!

If you have at least 2K, combine the two programs into one by deleting lines 205, 250, 260, 290, 210, and 220 and changing line 280 to read

```
280 LET T=L16
```

Your two REM statements combined now read conveniently

```

1 REM "INCOME AVERAGING"
2 REM "USING SCHEDULE Y"

```

If you get ambitious, you can rewrite this program with more steps such as to print out all the values needed to fill in a Schedule G form. You could even print it in similar order so it can be easily transposed onto the form.

THE END AND BEGINNING

This, the end of the book, is the beginning of your growth in the computer field. Whether you ever write another program or not, you at least know what it is all about. If you buy programs, you will know what you are getting for your money.

BASIC is no longer Greek to you. If you decide to upgrade to a more powerful machine such as the Sinclair Spectrum or the Commodore 64, you will find it very easy to convert into more sophisticated forms of BASIC—you would only have to learn how to use additional shortcuts!

The decisions in your future shall always reflect the knowledge and experience you have gained with your Timex/Sinclair 10000.

EPILOGUE: HISTORY OF THE TIMEX/SINCLAIR 1000

Now that you have experience with your computer and can appreciate its inventor's genius, you may wish to know how your computer evolved from Clive Sinclair's creation—the ZX80, ancestor to the Timex/Sinclair 1000.

The ZX80, grandfather to the Timex, was similar in size and offered in a white case. It first entered the British market in February 1980 with 1K of memory and selling for about \$200. A 16K-RAM expansion unit was contemplated for about \$700.

In the 1980s another British company, Micro Ace, entered the small-computer market with a computer resembling Sinclair's design to the point of almost being identical to it. Subsequent patent and copyright litigation resulted in allowing Micro Ace to market kits in the United States through their branch in California. Their kit was first offered with 2K memory in August 1980 at \$169.

In September 1980 Sinclair announced the development of a 16K-RAM pack for under \$100 using dynamic chips rather than the static type provided in the computer.

In January 1981, the ZX80 became available by mail order direct from Sinclair's U.S. office for \$199.

Then in April 1981, the development of the advanced ZX81 was announced in Great Britain. This unit involved a change from the 4K-ROM chip to

an 8K-ROM chip. To the user it meant many increased capabilities, including floating-point arithmetic (4K was limited to integer math only), addition of string arrays, and most new features now in the Timex. Although the design resulted from reducing some 20 integrated circuit chips down to just 4 major chips, Sinclair offered the 8K-ROM chip with a modified keyboard overlay for the ZX80 at \$39.

Concurrently, the computer publisher Creative Computing began publishing an excellent magazine devoted solely to the ZX-80 and subsequently also the ZX81, which of course is applicable to the Timex in all respects. If interested in this magazine, *SYNC*, write P.O. Box 789, Morristown, NJ 07960. This publication keeps you abreast of new products and software ideas for your computer.

By September 1981, interest in Britain had culminated in the world's first ZX Microfair. New products were displayed for the Sinclair, including a character generator, voice synthesizer, and color board—as well as the 16K-RAM pack.

Subsequently Sinclair introduced the printer in London. With 32 characters and 9 lines to the inch, it can print 50 characters per second on aluminized paper for \$99! Paper rolls of 64 feet cost about \$4.

In October 1981 the ZX81 was launched in the United States at \$149, or \$99 in kit form, as well as the 8K chip at \$39.

At the beginning of 1982 a second Micro Fair was held in London. Exhibits included even more progressive possibilities for the ZX81, including a Memopak of 64K, floppy disk storage unit, joystick, logic input/output devices, light pen, and voice recognition unit.

Meanwhile, Micro Ace has discontinued operations in the United States.

In April 1982, Timex—whose Scottish branch had been assembling the ZX81s—announced U.S. production of the Timex/Sinclair 1000 for distribution beginning in August 1982 at a price of \$99, and 16K expansion at only \$50. Timex has also promised eventual availability of a printer, Visi-Calc system, etc.

Clive Sinclair not only created the ZX80, he was also the inventor of the first pocket calculator in the mid-1960s. He pioneered digital watches and micro-TV sets. He founded Sinclair Research and developed the ZX80 in March 1979, and had this unit on the British market within a year. By the end of 1980 he had conceived the 8K BASIC.

Within two years over half-million Sinclair units were in use; Timex's distribution system will quadruple that amount. Sinclair has directed most of his recent efforts to producing a flat (3/8-inch) 2-inch monitor which would make his computer system really compact!

Based upon the rapid growth of the Sinclair and its auxiliary equipment, there appears to be great potential for the Timex unit. For example, new features already available by May 1982 included the 64K Memopak at \$160, a Byte-Back modem at \$99, relay control modules at \$59, an RS-232 port at \$60, and some sophisticated software programs in 16K machine language, including chess and adventure games at under \$25. The most valuable accessory to the beginner is the

printer and 16K memory, assuming he or she already has a television and a tape recorder.

What is on the Sinclair horizon? Expect to see his advanced color version called the Spectrum coupled with a flat video screen and disk storage all in a compact package. And probably at a very low price, too!

APPENDICES

APPENDIX A

Answers to Exercises

1.1 20 PRINT "NAME", "CITY"

Reason: Literals in a print statement require quotation marks.

or

20 PRINT N\$, C\$

Reason: String variables can only be identified with a letter and \$ symbol

1.2 20 INPUT N\$

22 INPUT C\$

24 INPUT S\$

The Timex will only accept one input variable at a time.

1.3 Yes. No restrictions.

1.4 Nothing.

1.5 Yes—in effect it says clear all variables, then go to line 30. Note, however, that Timex BASIC will not allow RUN 30–50.

- 1.6 The comma separates them into subsequent zones—that is, columns 0 and 16—which will appear as:

```
1      2
3
```

The semicolon leaves no space between them:

```
123
```

- 1.7 10 PRINT 1; "#####"; 2, "##"; 3
20 PRINT 1; TAB 6; 2; TAB 18; 3

- 1.8 No. It is, however, necessary in many other forms of BASIC.
1.9 A\$ has not been defined or INPUT. Error code is 2—undefined variable.
1.10 Result: ME will be printed at the center of the screen.
1.11 True. Note that with Timex INPUT statements calling for a string variable, the quotes are automatically provided.
1.12 Yes. A blank PRINT statement serves the purpose of skipping a line.
1.13 Your Timex can only handle statement numbers up to 9999.
1.14 False. There is no limit. You can essentially write a book.
1.15 True.
1.16 True.
1.17 It prints two column headings at columns 0 and 16.
1.18 True. (There are, however, special exceptions or tricks.)
1.19 10 REM: "SPARE PARTS"
20 PRINT "PART DESCRIPTION?"
25 INPUT D\$
30 PRINT "PART NUMBER?"
35 INPUT N\$
40 PRINT "PRICE?"
45 INPUT P\$
47 CLS
50 PRINT AT 10, 8; D\$; TAB 40; N\$; TAB 40; P\$
1.20 5 REM "NAME LIST"
9 REM STATEMENTS TO GATHER INFORMATION ON FIRST PERSON
10 PRINT "WHAT IS THE FIRST PERSON'S FIRST NAME?"
15 INPUT F\$
20 PRINT "WHAT IS HIS MIDDLE INITIAL?"
25 INPUT M\$
30 PRINT "HIS LAST NAME?"

```

35 INPUT T$
39 REM STATEMENTS TO GATHER INFO ON SECOND PERSON
40 PRINT "WHAT IS THE SECOND PERSON'S FIRST NAME?"
45 INPUT G$
50 PRINT "HIS MIDDLE INITIAL?"
55 INPUT N$
60 PRINT "HIS LAST NAME?"
65 INPUT U$
69 REM INFO ON THIRD PERSON
70 PRINT "WHAT IS THE THIRD PERSON'S FIRST NAME?"
75 INPUT H$
80 PRINT "HIS INITIAL?"
85 INPUT O$
90 PRINT "HIS LAST NAME?"
95 INPUT V$
99 REM CLEAR SCREEN
100 CLS
109 REM PRINT HEADING AND SPACES
110 PRINT "FIRST": TAB 10; "INITIAL"; TAB 20; "LAST"
120 PRINT
129 REM PRINT OUT NAMES UNDER HEADING
130 PRINT F$; TAB 13; M$; TAB 20; T$
135 PRINT G$; TAB 13; N$; TAB 20; U$
140 PRINT H$; TAB 13; O$; TAB 20; V$

```

Remember—your program may differ. The true test is whether it works. The best-written program is the shortest and most efficient.

- 2.1 Formula is $A = \frac{1}{2} bh$.

```

10 PRINT "INPUT H"
20 INPUT H
30 PRINT "INPUT B"
40 INPUT B
50 LET A = .5 * B * H
60 PRINT "AREA OF TRIANGLE IS #"; A

```

- 2.2 False. You can have as many nested loops as you want.

- 2.3 Only b will work as listed. The **RUN** in a will clear all your variables, destroying your results. The **CLEAR** in c has the same effect as **RUN** in a. Choice d is legitimate if you say **GO TO 60** instead of **RUN**. Choice b works because the arrow cursor will sit at the most recent line worked on. So you simply enter one digit before 59 as a blank. Then **EDIT** will automatically call out the next program line—in this case, 60. A shortcut! Don't call for line 60 this way because it would erase it!

- 2.4 Yes—as long as variables are defined somewhere in the program.
- 2.5 Loops are incorrectly nested. Switch around lines 40 and 50. *Note:* It works as is but gives an erroneous answer.
- 2.6 True. However, it cannot be keyed directly. It is shifted when in **K** mode. This also applies to **LPRINT**, **SLOW**, **FAST** and **LLIST** (keys S, D, F, and G).
- 2.7 False. Most keywords are both but not all. A command can be entered without a line number and will be operated immediately such as **LIST**, **RUN**, or **PRINT**. **INPUT** is the exception. The commands **RUN**, **LIST**, **CONT**, **CLEAR**, and **NEW** can be used in a program, although they would serve a limited purpose.
- 2.8 As lines are added when the screen is full, the top line gets knocked off. It is still there—imagine an invisible screen. Simply **LIST 0** and the listing will be displayed. **CONT** has no effect on a **LISTing**—only on results. To display more of the listing, request **LIST m**, where m is the last line number on the screen putting it on top. The best way to display the program in its entirety is with a printer.
- 2.9 B?QX because symbols are not allowed. 3TOM because it must begin with a letter. BACON AND EGGS is acceptable; however, it is identical to BACONANDEGGS!
- 2.10 Only one possible answer in each case—0!
- 2.11 Not quite, because of the variable R being undefined. But change R to a real number and it is perfectly OK.
- 2.12 10 INPUT S
20 PRINT "VOLUME OF CUBE WITH SIDE #"; S; "# IS"; S **3
- 2.13 10 FOR J = 1 TO 10
20 PRINT J
30 NEXT J
- 2.14 An idea: The counter could be used to keep one's score—for example, number of problems correctly answered.
- 2.15 All of them! Generally any keyword if it makes sense.
- 2.16 10 FOR N = 1 TO 9
20 PRINT N; "#";
30 NEXT N

Another approach:

```
10 FOR N = 29 TO 37
20 PRINT CHR$(N); "#";
30 NEXT N
```

2.17 LET C = 4 * (A + B)

2.18 LET K = (3 * (A + B + C) - P)/N **2 + 6

2.19 Nothing!

2.20 Yes (except quotes).

2.21 Look it up in your **TIMEX** handbook or type **PRINT CODE " "** entering the symbol in the quotes.

2.22 51—same as **CODE "N"**. The code function only gives the code of the first character of a string.

2.23 This doubles the amount of graphics available. One may want a dark background for a game.

2.24 It is important to distinguish types for versatility. Numbers can follow different rules than mere letters. The computer handles them differently. Numerics are looked at as numbers. Strings are treated as symbols. The symbol "2" is different from the number 2. It is possible to make a string have a numeric value or make a numeric into a string. The method and purpose for the Timex shall be described later.

2.25 Suggested solution:

```

05 LET J = 0
10 REM PRINT HEADING PLUS CALL FOR DATA AND START LOOP
20 PRINT AT 5, 0, "SALESMAN"; TAB 11; "I.D."; TAB 16; "SALES";
  TAB 25; "COMM"
25 FOR E = 1 TO 3
30 INPUT N$
40 INPUT I$
50 INPUT S1
60 INPUT S2
70 INPUT S3
80 INPUT S4
85 REM MAKE COMPUTATIONS
90 LET T = S1 + S2 + S3 + S4
100 IF T > 10000 THEN LET C = T * 0.15
110 IF T < 10000 THEN LET C = T * 0.10
120 LET J = J + C (Note: This is the accumulator)
125 REM PRINT OUT RESULTS
130 PRINT AT 6 + E, 0; N$; TAB 12; I$;
  TAB 17; T; TAB 25; C
135 NEXT E

```

```

140 REM PRINT TOTAL COMMISSIONS
145 PRINT AT 11, 1; "TOTAL COMMISSIONS PAID:
    # $"; J    (Colon is on Z key)

```

Note: The colon and period serve no special function other than punctuation.

2.26 Standard solution:

```

10 REM ESTABLISH INITIAL DATA-INTEREST, MONTH, ESTATE
20 LET M = 1
30 LET E = 100000
35 REM ESTABLISH QUARTERLY LOOP FOR INTEREST CALC
40 FOR J = 1 TO 3
50 LET E = E - 100
60 IF E < 10000 THEN GO TO 140
70 LET M = M + 1
80 NEXT J
85 REM THEN ADD INTEREST
90 LET E = E * 0.0525/4 + E
100 GO TO 40
140 PRINT "NUMBER OF MONTHS USED #"; M
150 PRINT "AMOUNT LEFT IS # $"; E
145 LET E = INT(E * 1000)/1000

```

Note: Line 145 rounds off decimals. Try it without line 145!

Answer: 120 months, \$978.52.

A more refined, tricky solution:

```

10 LET I = 0.0525
20 LET M = 1
30 LET E = 100000
40 FOR J = 1 TO 150
50 LET E = E - 100
60 IF M = 3 * INT(M/3) THEN LET E = E * (I/4 + 1)
70 LET M = M + 1
80 IF E < 10000 THEN GO TO 140
90 NEXT J
140 PRINT "PAYMENTS FOR #"; M; "# MONTHS LEFT YOU WITH
    #"; E

```

Answer: 121 months, \$991.37. Why is there a difference?

- 3.1 True—unless specified by the manufacturer. There is one minor difference among them which might affect some hardware. The ZX-81 and Timex have the connector board on the right side with all other input/output plugs on the sides. The Memopak 64K will therefore fit neater with these units, although it will work with the others.
- 3.2 False. Although it is true that software programs designed for the ZX-80 8K and Micro Ace 8K—including those in this book—are perfectly usable on the Timex 1000 or ZX-81, the reverse is not necessarily so. Some moving graphics programs written in BASIC require a **SLOW** computer speed—see keys D and F—not available on the ZX-80 8K. Micro Ace has developed a flicker-free board which adds this feature. Unless it is added to the ZX-80 8K, moving graphics will flicker unless programmed in machine language.
- 3.3 False. Don't worry, your Timex knows what you stuck on it!
- 3.4 Following is a tabulation of memory consumed comparing the two solutions to Exercise 2.26.

First Program:

	Line number	Characters	Single-stroke words	Real numbers
	20	3	1	1
	30	7	1	1
	40	4	2	2
	50	7	1	1
	60	9	3	2
	70	5	1	1
	80	1	1	0
	90	14	1	2
	100	2	1	1
	140	26	1	0
	<u>150</u>	<u>20</u>	<u>1</u>	<u>0</u>
Totals	11	Lines 98	14	11
factors:	$\times 5$	$\times 1$	$\times 1$	$\times 6$
Total bytes	55	98	14	66 = <u>233</u>

Note that if you include the **REM** statements, you would add another 126 bytes. But for comparison purposes to the second program, we won't consider it. Also line 145 is optional and not included.

Second Program:

	<u>Lines</u>	<u>Characters</u>	<u>Single strokes</u>	<u>Numbers</u>
	20, 30, 50, 70	22	4	4
	10	8	1	1
	40	6	2	2
	60	20	4	4
	80	9	3	2
	90	1	1	0
	<u>140</u>	<u>44</u>	<u>1</u>	<u>0</u>
Totals:	10 lines	110	16	13
	50 bytes	110 bytes	16 bytes	78 = <u>254</u>

One can see the first program is slightly better. There are ways to reduce memory consumption by combining lines, using characters instead of numbers, and limiting lengths of variable names and remark statements.

- 3.5 With 1K you have 1024 bytes less 125 bytes or 899 available. If you ran the program with **REM** statements, you would only have $899 - 233 - 126$ or 540 bytes. No problem! With 2K, you have 1564 bytes, and with 16K you have 15,900 bytes left. It becomes important with larger programs and especially when loading large arrays of numerical data.

4.1 It should be **LET Y = INT(RND * 6)**.

4.2 Given $P = 14$: therefore,

$Y64T = (2 * P) ** 64 - 1$

PRINT "DOLLARS = "; ((2 * 1) ** 64 - 1)/100

Answer: 1.8446744E * 17, which means \$184,467,440,000,000,000!

4.3 10 **FOR X = 1 TO 63**

20 **PLOT X, X**

30 **NEXT X**

4.4 10 **FOR X = 1 TO 63**

20 **PLOT X, LNX**

30 **NEXT X,**

4.5 10 **FOR T = 0 TO 60**

20 **LET A = T/30 * PI**

30 **LET SX = 31 + 9 * SIN A**

40 **LET SY = 21 + 9 * COS A**

50 **PLOT SX, SY**

60 **NEXT T**


```

4.6 10 FOR T = 0 TO 60
      20 LET A = T/30 * PI
      30 LET SX = 0 + 18 * SIN A
      40 LET SY = 21 + 18 * COS A
      50 PLOT SX, SY
      60 NEXT T

```

Note: Other ways are possible.

```

4.7 10 RAND
      20 FOR K = 1 TO 6
      30 LET Y = INT (RND * 36) + 1
      40 PRINT Y
      50 NEXT K

```

```

4.8 10 FOR N = 0 TO 63
      20 PLOT N, 22 + 10 * SIN(N/8 * PI)
      30 NEXT N

```

4.9 Load "SALARY SCALE" followed by this program:

```

10 LET I=S(3,5) * 2080
20 PRINT I

```

4.10 False. C is the simple variable, and only one of the other subscripted variables can coexist as a separate entity, not both.

4.11 True.

4.12 Figure-8 on its side.

4.13 Approximately 2 minutes. Hit any key.

4.14 Because it deals in the realm of strings only.

4.15 Yes, such as DIM S(A,B), but somewhere the variables must be defined before the computer will recognize any subscripted variables.

4.16 a. 2 d. 28
 b. 4 e. 14
 c. 28 f. -1

4.17 Nothing—it would plot and unplot immediately, and so because in effect there is no pause you will not see it.

4.18 Nothing. It will simply load the first program on tape which it encounters with any name.

4.19 No, the name must be in quotes.

4.20 The solution is presented twice—once with all the numbers so it is clear, and second to squeeze the program down to 322 bytes for 1K units. The display itself consumed 409 bytes alone in either case.

```

1 REM "CHESSBOARD"
10 FOR J = 2 TO 14 STEP 4
11 FOR K = 4 TO 16 STEP 4
12 FOR I = 8 TO 20 STEP 4
20 PRINT AT J, I; CHR$ 128;
    TAB I + 1; CHR$ 128;
    TAB I + 2; CHR$ 136;
    TAB I + 3; CHR$ 136
30 PRINT AT J + 1, I; CHR$ 128;
    TAB I + 1; CHR$ 128;
    TAB I + 2; CHR$ 136;
    TAB I + 3; CHR$ 136
40 PRINT AT K, I; CHR$ 136;
    TAB I + 1; CHR$ 136;
    TAB I + 2; CHR$ 128;
    TAB I + 3; CHR$ 128
50 PRINT AT K + 1, I; CHR$ 136;
    TAB I + 1; CHR$ 136;
    TAB I + 2; CHR$ 128;
    TAB I + 3; CHR$ 128
60 NEXT I
70 NEXT K
80 NEXT J

```

In the following, "shortened" form, note that a symbol with a line above it means Graphics mode, a line below means SHIFT:

```

1 REM "CHESSBOARD - 1 K RAM"
3 LET A = 1
4 LET B = A + A
5 LET C = B + A
6 LET D = C + A
10 FOR J = B TO 14 STEP D
11 FOR K = D TO D*D STEP D
12 FOR I = D + D TO 20 STEP D
20 PRINT AT J, I; "#";
    TAB I + A; "#";
    TAB I + B; "H";
    TAB I + C; "H"

```

```

30 PRINT AT J + 1, I; "  $\overline{\#}$  ";
   TAB I + A; "  $\overline{\#}$  ";
   TAB I + B; "  $\overline{H}$  ";
   TAB I + C: "  $\overline{H}$  "
40 PRINT AT K, I; "  $\overline{H}$  ";
   TAB I + A; "  $\overline{H}$  ";
   TAB I + B: "  $\overline{\#}$  ";
   TAB I + C: "  $\overline{\#}$  "
50 PRINT AT K + 1 "  $\overline{H}$  ";
   TAB I + A; "  $\overline{H}$  ";
   TAB I + B; "  $\overline{\#}$  ";
   TAB I + C; "  $\overline{\#}$  "
60 NEXT I
70 NEXT J
80 NEXT K

```

Notes: You can see how a little ingenuity can save you enough space to run this program on the ZX-81. In either case the lines do exactly the same thing. In the second program we eliminated as many real numbers as possible and replaced them with letters, saving 6 bytes each. Of course, lines 3 to 6 had to be added as initializers for the variables.

With the Timex, standard notation for the inverse symbols is the letter key with a dash over it. Hence, the inverse H or white H on black background is \overline{H} and is obtained by first changing the cursor to a **G**. To denote the graphic symbols we may have to *shift*. In that case a line below the letter is used: \overline{H} means get into Graphics mode—then depress **SHIFT H** and you will get the symbol for character number 136!

This program included three nested loops. The J loop prints out the lines with black squares first. The K loop does the white squares first. The L loop works within these loops to repeat the process for 8 chessboard columns or 16 Timex columns.

To see what it is actually doing, you can insert **GO TO** statements in the program and bypass lines to see what each line does. For example,

```
15 GO TO 40
```

will let you see just what line 40 does.

5.1 False. The correct value is 54. Here's how:

```

LEN "TODAY" = 5
STR$ 5 = "5"
"5" + "4" = "54"
VAL "54" = 54

```

To get 9 as a result, “4” in the original problem would have to be replaced by “+4”.

5.2 12

5

12 16

Explanation: Lines 15, 20, and 25 establish the value of the B\$ variable for the five locations, allowing up to five spaces for each. The values are the strings 4, 8, 12, 16, 20. Line 40 caused the third and fourth to be printed: 12 and 16 in two columns. Line 30 caused the value of the third to be printed, which is of course 3. The length of line 35 is printed as 5 because the dimension statement set aside five spaces. In reality, the strings are really not 4, 8, 12, 16, and 20 but 4#####, 8#####, 12#####, 16#####, and 20#####—in other words, the unseen spaces are in fact included and must be considered when playing with strings.

5.3 Nothing—syntax error. You cannot subtract strings. Actually, you can’t “add” them either; you really only “connect” them!

5.4 60 DEGREES.

5.5 Program a: SUPE

Program b: P

This is why you cannot mix A\$ as a variable name for a string as well as a subscripted variable. Only SUPE was printed because the dimension statement only allowed four spaces. The parentheses after a string variable name therefore is a slicer *unless* the string variable has been dimensioned.

5.6 Error 3—you cannot put in a substring larger than the string. Rewrite line 10 and reRUN:

```
10 LET B$ = "WE LOVE TO GO TO THE ZOO ###"
```

Now you will get:

```
WE LIKE TO GO TO THE MUSEUM
```

5.7 This undoubtedly a tricky application of what you learned, and it required some thought. REM statements are included for explanation only—don’t type them in.

```
5 REM ESTABLISH STRING ARRAY CONTAINING EACH MONTH
  SPELLED OUT PLUS A NUMERIC ARRAY CONSISTING OF THE
  NUMBER OF LETTERS IN EACH WORD.
```

```
10 DIM M$(12,9)
```

```
11 REM THE SECOND DIMENSION OF M$ ESTABLISHES
  THE MAXIMUM LENGTH WORD (SEPTEMBER)
```

```
12 DIM L(12)
```

```
15 FOR M = 1 TO 12
```

```
20 INPUT M$(M)
```

```

21 REM ENTER THE 12 MONTHS
22 INPUT L(M)
23 REM ENTER THE NUMBER OF CHARACTERS PER MONTH
25 NEXT M
28 REM INCLUDE A PRINTOUT TO ASK FOR THE MONTH BY NUM-
    BER (1 THROUGH 12)
30 PRINT "WHAT MONTH NUMBER?"
35 INPUT K
36 REM CLEAR THE SCREEN OF CLUTTER
37 CLS
39 REM NOW PRINT OUT THE RESULT
40 PRINT "THIS IS THE MONTH OF #"; M$ (K, 1 TO L(K));" "

```

Note the trick is in line 40 using a variable slicer L(K)! This slices off the unwanted spaces automatically provided by the dimension statement and prints out only the letters of the word based upon the stored value of the length of each word. This trick insures the period is always in the correct position!

Your solution may not be the same. In any case, **RUN** it and be amazed!

- 5.8 DIM D\$ (4,10,8) Four columns, 10 lines in each, maximum word is SATURDAY of 8 letters. D\$ (3, 2, 4 TO 6). Since it would be part of any day of the week, it would most likely be the word "DAY" since 5 of the 7 days are only 6 letters long. It could also be URD for SATURDAY and RSD for THURSDAY!
- 5.9 The condition is true if its value is *not* zero. Therefore, a and b are true; b is true because c is false—the NOT reverses it.
- 5.10 Here a and b become false and c and d become true.
- 5.11 A. X must be 3.
B. Z must be 0.
C. Y must be greater than 4.
D. False.
E. True.
- 5.12 IF A = 1 OR A = 2 THEN PRINT B\$
IF A > 0 AND A < 3 THEN PRINT B\$
IF A ≥ 1 AND A ≤ 2 THEN PRINT B\$
- 5.13 A is a numeric variable.
A1 is another numeric variable.
A\$ is a string variable.
A(I) is a subscripted numeric variable.
A\$(I) is a the Ith alphanumeric of a string variable.
"A" is a single alphanumeric.

5.14 The long way is to write 11 statements, such as

```
5 IF V = 1 THEN GO TO 100
10 IF V = 2 THEN GO TO 150      etc.
```

You can simplify by using appropriate line numbers:

```
10 IF V THEN GO TO V * 100
```

Where each line is numbered 100 through 1000.

5.15 $A = 15$ if $X = 2$

A = Whatever it was before if X is anything but 2. The computer evaluates the truth of the statement within parentheses.

5.16 a , d , and e ; b and c .

5.17 b and c .

5.18 Add the statements in the following lines 05, 55, 75, 76, 77, 80, and 85 and modify line 50:

```
05 LET C = 0
08 LET W = 0
10 PRINT "ODD OR EVEN?"
20 LET N = INT(RND * 100) + 1
30 INPUT G$
40 IF CODE G$ = 49 THEN GO TO 90
50 IF CODE (G$) = 52 - (INT(N/2) = N - N/2) * 10 THEN GO TO
    75
55 CLS
57 LET W = W + 1
59 PRINT W: "# WRONG AND"; C "CORRECT"
60 PRINT "NUMBER IS #"; N
70 GO TO 10
75 LET C = C + 1
76 CLS
77 PRINT C; "# CORRECT"
80 PRINT "NUMBER WAS #"; N
85 GO TO 10
90 LIST
```

5.19 45 IF (VAL X\$ < VAL Y\$ AND L = 22) OR ((VAL X\$ > 5 OR VAL Y\$ > 5) AND L = 23) THEN GO TO 30

5.20 This gives you the number of bytes in your program—which is, including line 145, 552 bytes. If you actually combined lines 45 and 47, as above, it should be 541 bytes.

5.21 The **CODE** "1" is 24; therefore, change line 40 to read

```
40 LET L = INT(RND * 4) + 21
```

To prevent division by 0, Y\$ must never be zero. Also, the integer of X/Y must be equal to X/Y. Hence add:

```
42 IF L = 24 AND VAL Y$ < > 0 AND VAL X$/VAL Y$ < > INT(VAL
    X$/VAL Y$) THEN GO TO 30
```

Note that this does have the detrimental effect of reducing the probability of a division problem to occur.

5.22 M + 1 or 4.

5.23 **GO TO JAIL.**

6.1 2 ** 32 — or over 2000 million addresses!

6.2 10 **PRINT "INPUT DECIMAL UP TO 255"**

```
20 INPUT D
```

```
30 LET C = 128
```

```
40 FOR B = 1 TO 8
```

```
50 IF D < C THEN GO TO 90
```

```
60 LET D = D - C
```

```
70 PRINT "1";
```

```
80 GO TO 100
```

```
90 PRINT "0";
```

```
100 LET C = C/2
```

```
110 NEXT B
```

Note: This only works on numbers which can be represented within 8 bits. A more complicated program would be needed for larger numbers.

6.3 To find it you need to

```
PEEK 16388 = 256 * PEEK 16389
```

You want to relocate it by 40 bytes—that is, leave some room at the top for machine code routines. So your objective address of a new RAMTOP is

```
PEEK 16388 = 256 * PEEK 16389 — 40
```

Now this address has to be poked into bytes 16388 and 16389 individually. Hence this must be converted to two numbers each less than 256.

You **POKE** into address 16389 the number you get when you divide by 256. The remainder is stored into address 16388.

Then you check it by **PEEK**ing the address storing RAMTOP.

In program format this would read:

```
5 INPUT D
10 LET A = PEEK 16388 + 256 * PEEK 16389 - D
20 LET B = INT (A/256).
30 LET C = A — 256 * B
40 POKE 16389, B
50 POKE 16388, C
60 PRINT PEEK 16388 + 256 * PEEK 16389
```

D is the amount of space you need. Try D = 40 the first time. Try D = 1500 if you have the T/S 1000. Write and **RUN** some programs and you will see what happens when you choke a memory.

6.4 *Program:* 57 bytes—5 bytes for line, 40 bytes for characters, and 12 bytes for two numbers.

Display: 63 bytes—19 ENTER, 10 SPACES, and 34 characters/spaces.

APPENDIX B

Video Display Chart

When designing a program's output or dealing with pixels, it is helpful to utilize a video display chart. The T/S 1000 is unique with its 24×32 display. Hence a chart designed exclusively for the T/S 1000 is shown here. Line and column numbers will help you with **AT** statements, and X and Y coordinates will simplify your **PLOT** statements. You may also use the form to write your programs.

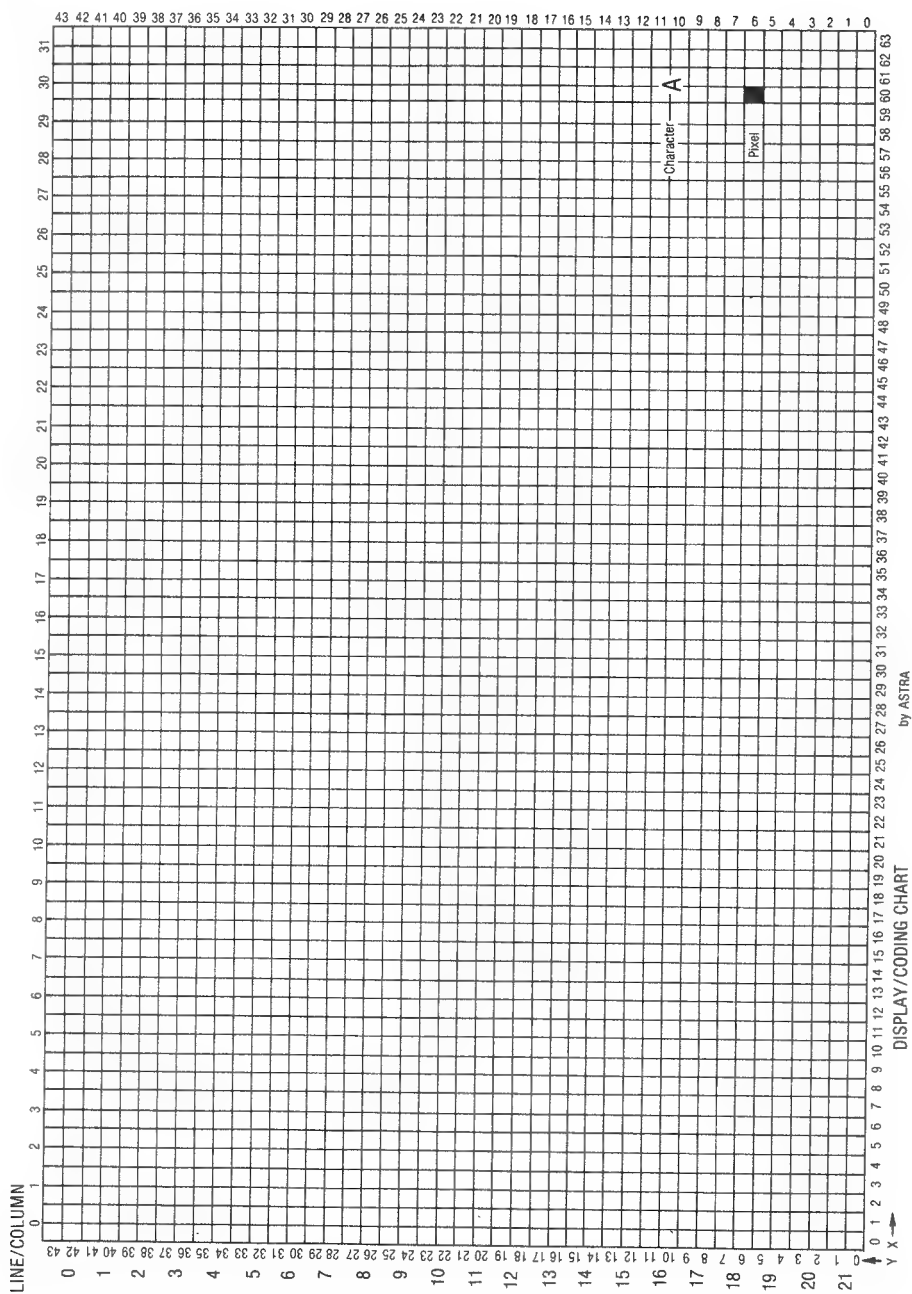
Charts can be obtained in lots of 20 for \$2.00 or 60 for \$5.00 from Astra Service, P.O. Box 234, Mt. Laurel, NJ 08054.

The User

Astra and the author are interested in *you*, the Timex user. Any comments regarding the Timex and/or this manual will be appreciated. If you have any programming/software questions, please send them to the above address with \$2.00 for a prompt response.

Please note in all correspondence the hardware you own—such as the Timex/Sinclair 1000, Sinclair ZX-81, Sinclair ZX-80 with 8K ROM, 16K Rampak, 64K Memopak, or other peripheral equipment. In this manner you can be notified of any manual corrections or future software for your T/S 1000.

VIDEO DISPLAY CHART



APPENDIX C: COMPUTER CODES

Timex Error Codes

- Ø—OK—no problem.
- 1—Error in the FOR-NEXT statement.
- 2—Undefined variable.
- 3—Variable subscript is out of range.
- 4—Insufficient memory.
- 5—No more room on the screen (use CONT).
- 6—Arithmetic overflow—beyond 10 ** 38.
- 7—Your RETURN statement didn't have a GOSUB.
- 8—INPUT cannot be used as a command.
- 9—Program stopped by STOP (use CONT).
- A—Function has invalid argument.
- B—Integer is out of range.
- C—VAL statement is invalid.
- D—You forced the program to stop using BREAK.
- F—SAVE needs a program name.

Useful Character Codes

<u>Character</u>	<u>Code</u>
Ø	28
1	29
2	30
3	31
4	32
5	33
6	34
7	35
8	36
9	37
A	38
B	39
C	40
D	41
E	42
F	43
G	44
H	45

<u>Character</u>	<u>Code</u>
I	46
J	47
K	48
L	49
M	50
N	51
O	52
P	53
Q	54
R	55
S	56
T	57
U	58
V	59
W	60
X	61
Y	62
Z	63
Space	0
+	21
-	22
*	23
/	24
**	216
Black-space	128
Graphics	1 through 10 128 through 138

APPENDIX D: FORMULAS

1. Program bytes consumed: **PRINT PEEK 16396 + 256* PEEK 16397-16509**
2. Relocating RAMTOP by "B" bytes:
LET A=PEEK 16388+256*PEEK16389-B
POKE 16389, INT (A/256)
POKE 16388,A-256*INT(A/256)
3. Drawing a circle:
PLOT X+B*SIN A,Y+BCOS A
Where B is the radius in pixels, A is an angle range between 0 and 2 PI radians, and X, Y are the center coordinates.

4. Annual cost of a loan—principal plus interest:

$$C = (P * I * (1 + I)^N) / ((1 + I)^N - 1)$$

Fill in any other formulas you find useful.

INDEX

A

ABS, 72
ACS, 71
alphabetical order, 100
AND, 102
argument, 68
arithmetic functions, 33
arrays, 82
arrow key, 12
art of programming, 3
ASN, 71
AT, 22-24
ATN, 71

B

bar chart, 80-81
BASIC, 1
binary system 121
bit, 62, 121
Boolean operators, 101

BREAK, 43
byte, 62, 120

C

Catch 'em, 146-47
CHR\$, 47
circle, 75-76
CLEAR, 32
clearing the screen, 10
clock, 76
CLS, 10, 18, 32
concatenation, 44
CODE, 47, 73
colon, 118
comma, 18
command, 2
computation order, 36
CONTINUE, 32
COPY, 119
cosine, 70

counter, 42
CPU, 62
cursor, 2

D

decimal system, 121
DELeTe, 8
DIMension, 82, 93
division, 33
dollar sign(\$), 92
Drawing, 145-46,

E

EDIT, 26
editing, 13, 25
electronic die, 68
ENTER, 9
EXP, 72

F

F cursor, 31
FAST, 76
Finance Plan, 157
flowchart, 48
flowchart symbols, 49
FOR. . .NEXT loop, 44
Fun loops, 46
FUNCTION, 2

G

G cursor, 31, 47
GOSUB, 151-52
GOTO, 20, 40
GRAPHICS, 47, 73
graphics mode, 47
Guessing Game, 79
Guess Me, 144-45

H

Hangman, 140-42
happy face, 130
hardcopy, 62
hardware, 61
hexadecimal system 121

I

IF, 42
IF. . .THEN, 100
Income Tax Averaging, 159-62
INKEY\$, 78
INPUT, 15-16
INT, 68
integer, 45
inverse video, 47
Investment return, 156-57

K

K, 62
K cursor, 6, 8
Knockout, 148-49

L

L cursor, 7
LEN, 94
LET, 36, 104
levels, 3
line number, 2, 13
LIST, 15
LLIST, 119
LN, 72
LOAD, 87
logic, 99
logical operators, 99, 105
loops, 42
Lottery Number, 152-53
LPRINT, 119

M

machine language, 62, 119
Math program, 114
memory, 32, 63, 120-23
microprocessor(Z80A), 62
minus(-), 33
mode, 3
modem, 64
mortgage loan, 157-58
multiplication(*), 33

N

nested, 34
NEW, 10, 32

NOT, 101, 104
 numeric arrays, 82
 numeric variables, 36
 Numerology, 143

O

Odd or Even , 105
 OR, 103

P

Parentheses, 34
 PAUSE, 76
 PEEK, 119, 124
 period, 118
 PI, 38
 pixel, 73
 PLOT, 70, 73
 plus(+), 33
 POKE, 76, 119, 127
 power(**), 35
 PRINT, 8, 106

Q

quote, 118

R

radian, 71
 RAM, 62
 RAMTOP, 122-24
 RAND, 68
 randomizing, 67
 REM, 27
 remark, 27
 RETURN, 152
 RND, 68
 ROM, 62
 rounding, 68-69
 RUN, 14-15

S

S cursor, 11
 SAVE, 87
 SCROLL, 77

semicolon, 21
 Sinclair, Clive, 164
 sine curve, 70
 SLC, 62
 slicing, 97-98
 Slot Machine, 147-48
 SLOW, 65
 softcopy, 62
 software, 61, 64
 sorting, 154-56
 SPACE, 7, 11
 SQR, 72
 statement, 2
 STEP, 44
 STOP, 32
 string array, 93
 string variables, 92
 STR\$, 93
 subroutine, 152
 substrings, 96-97
 STR\$, 93
 syntax, 1

T

TAB, 22, 23
 tangent, 71
 TO, 97
 trig functions, 69
 truth table, 101-3

U

UNPLOT, 73

V

VAL, 93-94
 variables, 15

W

word processor, 78
 working area, 9
 ZX80, 65, 164-65
 ZX81, 65, 165
 255, 47, 63, 122
 32767, 76
 65536, 68, 120, 122

Now that you own the most popular, most affordable computer—the Timex/Sinclair 1000 (or 2X81 or 2X80-8K)—on the market today, this book is the next investment you should make.

Written in an informal, easy-to-absorb style that can be readily grasped even by those with no previous programming experience, **Programming Your Timex/Sinclair 1000 in BASIC** is perfect for first-time computer users looking for a low-cost introduction to computers and their applications. Here you learn by doing as you work out each program. Each level incorporates different teaching techniques as you advance to higher levels of understanding, each of which includes:

- * new vocabulary with definitions and explanations of syntax
- * a step-by-step sample program using the new vocabulary
- * an interest stimulator—a short fancy program for you to run
- * practice programs
- * exercise problems with answers (in the appendix), including programs similar to the text, totally new programs, and poorly written programs (to show how **not** to program)
- * examples of errors and how to prevent/solve them
- * a summary of what you have learned

Whether you want to learn how to write your own programs for profit—or you want to make more intelligent decisions when buying “canned” programs—this hands-on approach will provide you with the skills and working knowledge needed. See how easy, fun, and profitable computing can be when you start **Programming Your Timex/Sinclair 1000 in BASIC**.

Mario Eisenbacher received a B.S. from Rensselaer Polytechnic Institute and an MBA from Widener College. He is currently a project engineer at Westinghouse Electric's combustion Turbine Division in Philadelphia. He has also completed **Programming Your Commodore 64 in BASIC** to be published in Fall 1983 by Prentice Hall, Inc.

PRENTICE-HALL, Inc., Englewood Cliffs, New Jersey 07632

Cover design by Hal Siegel



ISBN 0-13-729863-3